INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI®

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Replace a NULL Pointer by a Null Object Using Object-Oriented Method

Zhilin Li

A Report
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements for
The Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

March 1998

Abstract

Replace a NULL Pointer by a Null Object Using Object-Oriented Method

Zhilin Li

Pointers are one of the most powerful and flexible features in C/C++, but they are also one of the most dangerous ones to use. Safe programming practice requires a pointer to be initialized to NULL once declared. However a NULL pointer does not solve the problem completely. Accessing a NULL pointer may crash the application without offering any useful clues to the real problem. In Object-Oriented design, we can view a NULL pointer as a special kind of object with special behaviors and implement a null object to replace the NULL pointer. Accessing a null object will not crash the application, instead it can perform meaningful operations or display more helpful error messages. In this project, we implement a binary search tree, an AVL tree, a singly linked list using both the traditional NULL pointer method and the Object-Oriented null object method. We then compared the two methods in terms of safety, coding complexity, and time/space efficiency. The implementation language is C++.

Acknowledgments

The author would like to thank Dr. Peter Grogono who carefully supervised the project and offered many valuable opinions. Without his help, the project would not have been completed in so short a time.

Table of Contents

List of Figures.....VIII

List of Tables.....IX

List of Code Listings.....X

List of Test Results.....XII

1 INTRODUCTION..... 1

1.1 Why Do We Need Pointers? 1

1.2 Problems with Pointers..... 2

1.3 NULL Pointer Problem 4

1.4 Using a Null Object to Replace a NULL pointer 5

2 BACKGROUND 7

3 NULL POINTERS AND NULL OBJECTS 10

3.1 Implementing a Binary Search Tree 11

3.1.1 Null Node Method 12

3.1.1.1 Implementing LNullBTreeNode 13

3.1.1.2 Implementing LFullBTreeNode..... 14

3.1.1.3 Implementing LBTree..... 19

3.1.2 NULL Pointer Method..... 20

3.2 Implementing a Template Binary Search Tree 22

3.2.1 Null Node Method 23

3.2.2 NULL Pointer Method 24

3.3 Implementing an AVL Tree 25

3.3.1 Null Node Method 27

3.3.2 NULL Pointer Method 30

3.4 Implementing a Singly Linked List 31

3.4.1 Null Node Method 32

3.4.2 NULL Pointer Method 33

4 EXPERIMENTAL DESIGN 34

4.1 Test Correctness 34

4.2 Comparing Time and Space Efficiency 37

4.2.1 Calculating Time 38

4.2.2 Calculating Space 38

4.2.3 Comparing Time and Space Efficiency 38

5 EXPERIMENTAL RESULTS 40

5.1 Build Time Efficiency 40

5.2 Access Time Efficiency 43

5.3 Space Efficiency 44

6 CONCLUSIONS 50

7 BIBLIOGRAPHY 52

APPENDIX 53

List of Figures

FIGURE 1 : CLASS HIERARCHY TO REPRESENT A TREE NODE 5
FIGURE 2: A BINARY SEARCH TREE EXAMPLE 11
FIGURE 3: AN AVL TREE EXAMPLE 25
FIGURE 4: OBJECT DIAGRAM OF IMPLEMENTING AN AVL TREE 27
FIGURE 5: TEST RESULT OF BINARY SEARCH TREE OF INTEGERS 45
FIGURE 6: TEST RESULT OF TEMPLATE BINARY SEARCH TREE OF DOUBLES 46
FIGURE 7: TEST RESULT OF TEMPLATE BINARY SEARCH TREE OF INTEGERS 47
FIGURE 8: TEST RESULT OF AVL TREE OF INTEGERS 48
FIGURE 9: TEST RESULT OF SINGLY LINKED LIST OF INTEGERS 49

List of Tables

TABLE 1: COMPARING BUILD TIME OF THE TWO METHODS 40

TABLE 2: COMPARING ACCESS TIME OF THE TWO METHODS 43

List of Code Listings

CODE LISTING 1: *LBTREENODE.H* 53

CODE LISTING 2: *LBTREENODE.CPP* 57

CODE LISTING 3: *LBTREE.H* 64

CODE LISTING 4: *LBTREE.CPP* 66

CODE LISTING 5: *LPTRBTREENODE.H* 69

CODE LISTING 6: *LPTRBTREENODE.CPP* 71

CODE LISTING 7: *LPTRBTREE.H* 78

CODE LISTING 8: *LPTRBTREE.CPP* 80

CODE LISTING 9: *LTMPLBTREENODE.H* 84

CODE LISTING 10: *LTMPLBTREE.H* 94

CODE LISTING 11: *LTMPLPTRBTREENODE.H* 97

CODE LISTING 12: *LTMPLPTRBTREE.H* 105

CODE LISTING 13: *LAVLBTREENODE.H* 109

CODE LISTING 14: *LAVLBTREENODE.CPP* 111

CODE LISTING 15: *LTMPLSTACK.H* 119

CODE LISTING 16: *LPTRAVLBTREENODE.H* 122

CODE LISTING 17: *LPTRAVLBTREENODE.CPP* 123

CODE LISTING 18: *LTMPLPTRSTACK.H* 131

CODE LISTING 19: *LSLISTNODE.H* 134

CODE LISTING 20: *LSLISTNODE.CPP* 137

CODE LISTING 21: *LSLIST.H* 143

CODE LISTING 22: *LSLIST.CPP* 144

CODE LISTING 23: *LPTRSLISTNODE.H* 146

CODE LISTING 24: *LPTRSLISTNODE.CPP* 148

CODE LISTING 25: *LPTRSLIST.H* 153

CODE LISTING 26: *LPTRSLIST.CPP* 154

CODE LISTING 27: *LSTATISTIC.H* 157

CODE LISTING 28: *LSTATISTIC.CPP* 160

CODE LISTING 29: *LRANDOMGEN.H* 162

CODE LISTING 30: *LTYPES.H* 163

CODE LISTING 31: *MAIN.CPP* 164

List of Test Results

TEST RESULT 1: *TESTTREE.TXT* 179

TEST RESULT 2: *TESTPTRTREE.TXT* 182

TEST RESULT 3: *TESTINTTMPLTREE.TXT* 185

TEST RESULT 4: *TESTINTTMPLPTRTREE.TXT* 188

TEST RESULT 5: *TESTAVLTREE.TXT* 191

TEST RESULT 6: *TESTPTRAVLTREE.TXT* 202

TEST RESULT 7: *TESTSLIST.TXT* 213

TEST RESULT 8: *TESTPTRSLIST.TXT* 216

1 Introduction

It is folklore among C/C++ programmers that "If you don't understand pointers, then you do not know how to program C/C++". While pointers are one of the strongest features of C/C++, they are also one of the most dangerous ones. Great efforts have been made by programmers to reduce the danger of pointers.

This project is an attempt to provide an alternative solution to reduce the danger associated with the pointers using Object-Oriented methods. We view a NULL pointer as an object with special behaviors and use a fully defined null object to replace a NULL pointer. We implement a binary search tree, an AVL tree and a singly linked list using both the traditional NULL pointer method and the Object-Oriented null object method. We also compare of the two methods in terms of safety, coding complexity, and time/space efficiency.

1.1 Why Do We Need Pointers?

A pointer is the address of an object in memory. The main functionality of pointers is for dynamic memory allocation which allows us to allocate memory of an object at run time. Dynamic memory allocation is very useful since most times we do not know how large a tree, a list or an array should be at compile time, and we need to allocate memory dynamically at run time. When an object is dynamically allocated at run time, the

memory address of the object, i.e. the pointer to the object, is returned for later reference to the object.

Pointers can also be used for passing parameters to functions by their addresses so that functions can modify the values of the calling arguments, or improving the efficiency of certain routines. For example, accessing an array using a pointer is faster than using an index. In addition, to achieve the C++ polymorphism feature -- one interface and multiple methods -- the virtual functions must be invoked through a base class pointer or reference.

1.2 Problems with Pointers

Pointers give programmers tremendous power and are necessary for many programs. At the same time, when a pointer accidentally contains a wrong value, it can be the most difficult bug to find. An erroneous pointer is difficult to find because the pointer, itself, is not the problem, the problem is that each time you perform an operation using the bad pointer, you may be reading or writing to some unknown piece of memory. The error may not show up until later execution of your program and can lead you to look for the bug in the wrong place. There may be little or no evidence to suggest that the pointer is the original cause of the problem. Sometimes, the pointer problem can be even trickier. Consider the following case in C++:

```
class C
{
public:
  void f();
  ...
private:
  int m_x;
};
```

Somewhere if we declared a pointer to *C* and used it to call function *f()*:

```
{
  C *p;
  // ...
  // Several pages of code
  // ...
  p->f();
}
```

Though *p* is a uninitialized pointer, if *f()* does not access any member variable in class *C*, then the statement *p->f()* will not cause any problem. But if sometime later, we add a code in *f()* like

```
void C::f()
{
  m_x= 5;
  ...
}
```

Then statement *p->f()* will crash. The following explains why:

When *f()* is called, "this" pointer is set to p. Since *f()* is not a virtual function, the compiler knows its address and simply sets "*this=p*" and calls *f()*. Therefore nothing is wrong until *f()* tries to access a member variable in which case

```
this->m_x= 5;
```

becomes

```
p->m_x= 5;
```

and the program crashes.

This case actually happened in a Montreal-based software company. By the time the error showed up, it was very difficult to guess p is a uninitialized pointer since dereferencing it did not cause a problem before.

1.3 NULL Pointer Problem

NULL pointer is a special value defined in C/C++ which can be assigned to each pointer type. It is guaranteed to compare unequal to a pointer to any object or function. Implementation of trees and linked lists use NULL pointers to mark an empty node in a tree and the end of a list.

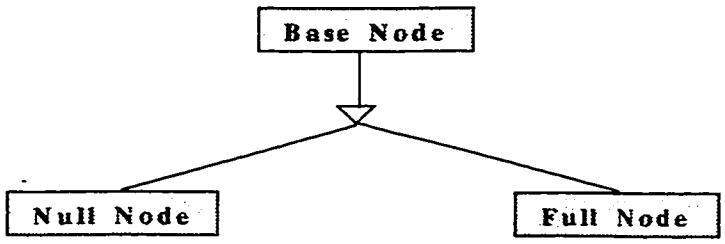
Safe programming practice requires that whenever you declare a pointer type variable, you should initialize it to NULL; and whenever you access a pointer, you should check that the pointer is not equal to NULL. In this way, we can avoid accessing a uninitialized pointer.

However the NULL pointer is not an ideal solution to the pointer safety problem. First, checking for NULL before dereferencing a pointer requires time overhead. Second, it is up to each individual programmer to initialize and check for the pointer value before using it. This is not truly safe. Third, when accessing a NULL pointer, the program may just crash instead of handling the error more gracefully. Different systems generate different errors when accessing a NULL pointer. For example, under Windows NT, accessing a NULL pointer crashes the application and displays a message like this *'The instruction at "0x00401007" referenced memory at "0x00000000". The memory could not be "written"*. In a Sun workstation, accessing a NULL pointer raises a segmentation fault.

1.4 Using a Null Object to Replace a NULL pointer

The most common use of NULL pointers is to represent the end of a list and the empty nodes of a tree. But logically NULL pointers are not required in Object-Oriented (OO) programming. Using a binary search tree as an example, there are two possibilities for a tree node: a node is either a *full* tree node, with data and pointers to subtrees, or an *empty* or *null* node, with no data at all. Therefore, the correct OO representation of a tree node uses a base class and two derived classes.

Figure 1 : Class Hierarchy to Represent a Tree node



In the **Base Node** class, we define all the possible operations that can be applied to a tree node as pure virtual functions. Then, the **Null Node** class and the **Full Node** class can implement the operations differently. For an operation that is not meaningful for a null node to perform, such as outputting the data at the null node, the node can display an error message, throw an exception or handle the case more gracefully. In such way, we can implement a tree or other data structures without using NULL pointers.

The advantages of the null node method are:

- Eliminate the danger of dereferencing a NULL pointer.
- Eliminate the requirements of checking for NULL before accessing a pointer.

In this project, we implement a binary search tree of integers, a template binary search tree, a height-balanced tree of integers and a linked list of integers using both the NULL pointer and the null node methods. We then compared the two methods in terms of safety, coding complexity, and time/space efficiency.

We choose C++ as the implementation language in this project. But other Object-Oriented programming languages should also support the null node method.

The C++ compiler is Microsoft Visual C++ 5.0. The tests are done under Windows NT 4.0 in a Pentium 100MHz with 32 MB memory.

2 Background

The most significant result of finding a way to use pointers safely is the **auto_ptr** template class which has become an official part of the C++ Standard Template Library (STL). There are several variations of the **auto_ptr** template class [[3], [4], [5], [6]]. This section will give a brief introduction of **auto_ptr**.

The **auto_ptr** template class describes an object that stores a pointer to an allocated object of type **T**. It acts much like a normal C++ pointer with improved safety:

- The default constructor initializes from NULL, therefore the stored pointer must either be NULL or designate an object allocated by a *new* expression. This reduces the chance of dereferencing a dangling or uninitialized pointer.
- Since we always use operators *->()* and **()* to access a pointer, the **auto_ptr** template class overloaded the two operators. The current implementation of the STL **auto_ptr** class did not have a check of NULL inside the operator functions, but it is possible to add the check, and throw an exception if the stored pointer is NULL.
- An auto_ptr object stores an ownership indicator. An object constructed with a non-NULL pointer owns the pointer. It transfers ownership if its stored value is assigned to another object. The destructor for the auto_ptr object deletes the allocated object if it owns it. Hence, an object of class **auto_ptr** ensures that an allocated object is automatically deleted when control leaves a block, even via a thrown exception. This eliminates a prime opportunity for memory leakage.

The definition of the **auto_ptr** template class is as follows[1]:

```
#define _THROW0()   throw ()

template<class _Ty>
 class auto_ptr {
public:
  typedef _Ty element_type;
  explicit auto_ptr(_Ty *_P = 0) _THROW0()
    : _Owns(_P != 0), _Ptr(_P) {}
  auto_ptr(const auto_ptr<_Ty>& _Y) _THROW0()
    : _Owns(_Y._Owns), _Ptr(_Y.release()) {}
  auto_ptr<_Ty>& operator=(const auto_ptr<_Ty>& _Y) _THROW0()
    {if (_Ptr != _Y.get())
      {if (_Owns)
         delete _Ptr;
       _Owns = _Y._Owns;
       _Ptr = _Y.release(); }
    else if (_Y._Owns)
      _Owns = true;
    return (*this); }
  ~auto_ptr()
    {if (_Owns) delete _Ptr; }
  _Ty& operator*() const _THROW0()
    {return (*get()); }
  _Ty *operator->() const _THROW0()
    {return (get()); }
  _Ty *get() const _THROW0()
    {return (_Ptr); }
  _Ty *release() const _THROW0()
    {((auto_ptr<_Ty> *)this)->_Owns = false;
    return (_Ptr); }
private:
  bool _Owns;
  _Ty *_Ptr;
  };
```

THROW0 is defined as *throw()* which indicates that a function does not throw any exception.

---

[1] Copyright (c) 1995 by P.J. Plauger. ALL RIGHTS RESERVED.

The **auto_ptr** template class is like a wrapper around a normal pointer. Comparing to the Object-Oriented null node method implemented in this project, it has two main advantages: (1) It is more general. It can represent any type of pointers. (2) It forces an initialization to NULL inside the constructor and a deletion of the privately pointed and owned object in the destructor.

However the **auto_ptr** template class does not solve the problem of accessing a NULL pointer. Although we can add a check of NULL inside the *() and ->() operators, it forces the check of NULL on every reference of a pointer. This is not what we want since a pointer is not NULL in most cases.

The Object-Oriented null node method provides a particular solution to the problem of dereferencing a NULL pointer. It is not an attempt to replace the functionality of the **auto_ptr** template class, but an effort to show how the Object-Oriented idea can be applied in improving the coding safety in the programming practice.

3 NULL Pointers and Null Objects

As we discussed in section 1.4, we can use a null node to replace a NULL pointer. In this section, we will give the implementation details of the null node method. We selected three data structures to implement: a binary search tree, an AVL tree and a singly linked list. The reasons we choose the above structures are: (1) A linked list and a binary search tree require pointers for allocating their nodes dynamically. (2) A binary search tree has a large amount of empty nodes. It is good for comparing the space/time efficiency after replacing a NULL pointer by a null node. (3) An AVL binary search tree is more complex, we can use it to test whether the null node method will work in a more complex situation and what the time and space efficiency will be in such situation. It can also demonstrate the mechanism of further subclassing in the node class hierarchy we defined in the null node method. (4) We also considered other data structures, but found that the implementation of these data structures is not as significant as the implementation of the data structures we have chosen. For example, the implementation of a doubly linked list is similar to a singly linked list. The implementation of a hash table or a graph can use a single or doubly linked list.

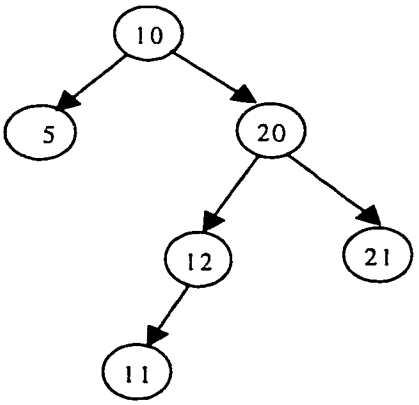
To simplify the problem, we assume that the data item in the trees and lists is an integer. But in order to show how the implementation can be generalized to work with any data type, we also implement a template binary search tree. To avoid conflicting with STL

(Standard Template Library), MFC (Microsoft Foundation Class) and other C++ class libraries, all the class names in this project start with the capital letter ***L***.

3.1 Implementing a Binary Search Tree

A **tree structure** is used to store data in sorted order. It consists of *nodes* that contain data. *Nodes* have links to other nodes. Any tree has a single *root node* to which all other nodes are linked. A node data must have a key which can be compared and sorted. A **binary search tree** is the simplest type of a tree structure. Each node in a binary search tree contains data and two links. The left link connects to all nodes with lesser data values; and the right link connects to all nodes with greater data values. One data is greater or lesser than another data can be determined by comparing the key values of the two data. The links are usually implemented by pointers. When a node does not have lesser or greater nodes, the left or the right link is a NULL pointer. Here is an example of a binary search tree with integer values:

Figure 2: A Binary Search Tree Example



Typical binary search tree operations are: insert a node into the tree, delete a node from the tree, traverse the tree and search a node in the tree. Since each tree node already defined its left and right subtrees, it can be viewed as a valid binary search tree itself. Base on this, we can implement the tree operations in each node. Only when implementing the operations at the node level, we can define the class hierarchy as in Figure 1 (page 5) and allow a full node and a null node to implement these operations differently. Otherwise if we implement the operations at the tree level, the tree must check for the null node before the operation – it is the same as check for NULL.

3.1.1 Null Node Method

In this implementation, no NULL pointer is used. Four classes are defined:

- LBaseBTreeNode. This is the base class of LNullBTreeNode and LFullBTreeNode. It defines all the node operations as the pure virtual functions. Pure virtual functions specify a consistent interface that all the subclasses must implement.
- LNullBTreeNode. It contains no data and represents an empty tree node.
- LFullBTreeNode. It represents a full tree node. It contains data and links (pointers) to the left and the right subtrees.
- LBTree. It acts like a wrapper around a LBaseBTreeNode and provides an easy-to-use interface. It contains a pointer to the LBaseBTreeNode as the tree root.

Here we give the definition of class LBaseBTreeNode since it defines the node operations that will be implemented by both LNullBTreeNode and LFullBTreeNode. For

more implementation details of the classes, please refer to *LBTreeNode.h* in page 53, *LBTreeNode.cpp* in page 57, *LBTree.h* in page 64, and *LBTree.cpp* in page 66.

```
class LBaseBTreeNode
{
public:
    virtual LBaseBTreeNode* insert(const int data) = 0;
    virtual LBaseBTreeNode* remove(const int data, BOOL &bDeleteThis, BOOL & bRemoveOK) = 0;
    virtual BOOL search(const int data) = 0;
    virtual void traverse(void) const = 0;
    virtual void traverse(ofstream& ofs) const = 0;
    virtual BOOL isNull(void) const = 0;
    virtual int getSelfData(void) const = 0;
    virtual void linkLeftSubtree(LBaseBTreeNode* pLeft) = 0;
    virtual void linkRightSubtree(LBaseBTreeNode* pRight) = 0;
    virtual const LBaseBTreeNode* getLeftSubtree(void) const = 0;
    virtual const LBaseBTreeNode* getRightSubtree(void) const = 0;
    virtual LBaseBTreeNode* getLeftSubtree(void) = 0;
    virtual LBaseBTreeNode* getRightSubtree(void) = 0;
    virtual void outputSelfData(void) const = 0;
    virtual void outputSelfData(ofstream& ofs) const = 0;
    virtual getHeight(void) const {return -1;}
    virtual long space(void) const = 0;
};
```

3.1.1.1 Implementing LNullBTreeNode

When an operation is not meaningful for a null tree node to perform, it can throw an exception or display an error message. In this implementation, for simplification, we just display an error message to the standard output. For example, function *getSelfData()* returns the data stored in the tree node. Since a LNullBTreeNode object does not contain any data, we write the function as follows:

```
int getSelfData(void) const
{
  cout << "ERROR: cannot get data from a NULL binary tree node." << endl; return -1;}
}
```

A null tree node can also implement the operations with a default response. For example, in function *outputSelfData()*, it prints "NIL" to the output to indicate an null node is

reached. In function *search()*, it simply returns FALSE as it does not contain the data to be searched.

When we insert data into a null tree node, the null tree node will become a full tree node. How could we convert a null tree node (LNullBTreeNode) to a full tree node (LFullBTreeNode) at run time? C++ does not provide a mechanism to do this. We solve the problem by letting the *insert()* function return a pointer to LBaseBTreeNode. Here is the function definition:

```
LBaseBTreeNode* insert
  (const int data)
{
  return new LFullBTreeNode(data);
}
```

The returned pointer, which points to the newly created LFullBTreeNode, will be used as the new root of the binary search tree whose root was the null tree node before.

3.1.1.2 Implementing LFullBTreeNode

In a binary search tree, there are large amounts of null nodes. In fact, there are always more than half of the total nodes are null nodes. There is no difference among these null nodes. If we represent each null node using a new LNullBTreeNode, it is a big waste of memory space. In fact, we only need to allocate one LNullBTreeNode and let all the null links point to the same null node. We do this by defining a static member variable inside class LFullBTreeNode:

```
static LNullBTReeNode m_nullNode_S;
```

C++ requires the initialization of *m_nullNode_S* to be done outside the class definition, usually in the implementation file:

```
LNullBTreeNode LFullBTreeNode::m_nullNode_S;
```

The above statement tells the compiler to allocate memory for the static null node. In C++, a static member variable is allocated only once when the application starts and is freed when the application terminates. We can use this static member variable *LFullBTreeNode::m_nullNode_S* to represent all the null nodes in the binary search trees. I.e. all the null nodes in all the binary search trees in an application will point to this single static null node. When a LFuliBTreeNode object is newly constructed, the constructor initializes its left and right links to point to the static null node.

When implementing operations for a full tree node, such as *insert()*, *remove()*, *search()* and *traverse()*, we fully utilize the dynamic binding and dispatching feature of virtual functions. We also use recursive functions to simplify the coding. Here we will use functions *insert()* and *remove()* as an example to present the implementation details. Let us first look at function *LFullBTreeNode::insert()*.

```
LBaseBTreeNode* LFullBTreeNode::insert
  (const int data)  // new data item to be inserted
{
  if (data < m_data)
  {
    m_pLeft= m_pLeft->insert(data); // recursive function call
  }
  else if (data > m_data)
  {
    m_pRight= m_pRight->insert(data); // recursive function call
  }
  // else {m_data = data} we don't allow duplication of data
  Return this;
}
```

We assume that there is no duplication of data in the binary search tree. Function *insert()* returns a pointer to LBaseBTreeNode which is the new root of the binary search tree represented by the tree node after insertion. When new data is inserted into the binary search tree, if the data is less than the data contained in the full node, it should be inserted into the left subtree, so we recursively call *insert()* function using *m_pLeft* (pointer to the left subtree). Similarly, if the data is greater than the data contained in the full node, we recursively call *insert()* function *m_pRight* (pointer to the right subtree). Since *insert()* is a virtual function defined in the base class of LFullBTreeNode and LNullBTreeNode, by dynamic binding and dispatching, if *m_pLeft* or *m_pRight* points to a LFullBTreeNode, then *LFullBTreeNode::insert()* will be called recursively; if *m_pLeft* or *m_pRight* points to the static null node *LFullBTreeNode::m_nullNode_S*, then *LNullBTreeNode::insert()* will be called and a new full node containing the new data will be created and linked to the binary search tree.

Now let us look at function *LFullBTreeNode::remove()*.

```
LBaseBTreeNode* LFullBTreeNode::remove
  (const int    data,  // data to be removed
   BOOL&  bDeleteThis, // TRUE if this node itself need to be deleted.
   BOOL&  bRemoveOK)  // TRUE if we sucessfully remove the data item from the tree
{

  LBaseBTreeNode *pNewRoot;

  bDeleteThis= FALSE;
  bRemoveOK= FALSE;

  if( data == m_data)
  {
    // ----------------------------------------
    // This node is the node which contains the
    // data item to be removed. Set bDeleteThis to
    // TRUE and bRemoveOK to TRUE.
    // ----------------------------------------
    bDeleteThis= TRUE;
    bRemoveOK= TRUE;
```

```
    if(m_pLeft->isNull())
    {
      pNewRoot= m_pRight; // new root is the right subtree.
    }
    else if(m_pRight->isNull())
    {
      pNewRoot= m_pLeft; // new root is the left subtree.
    }
    else
    {
      // ---------------------------------
      // None of the left and right subtree
      // is Null, we should replace the root
      // with the smallest node of the right
      // subtree.
      // ---------------------------------

      LFullBTreeNode* pNewThisNode;

      pNewThisNode= ((LFullBTreeNode*) m_pRight)->detachSmallestNode();

      // Connect the new root with the left subtree.
      pNewThisNode->linkLeftSubtree(m_pLeft);

      pNewRoot= pNewThisNode;
    }
  }
  else if( data < m_data)
  {
    BOOL bDeleteLeft;
    LBaseBTreeNode* pOldLeft= m_pLeft;

    m_pLeft= m_pLeft->remove(data, bDeleteLeft, bRemoveOK);
    if(bDeleteLeft)
    {
      delete pOldLeft;
    }

    // Data item is removed from the left subtree. The tree
    // root is still this node.
    pNewRoot= this;
  }
  else if(data > m_data)
  {
    BOOL bDeleteRight;
    LBaseBTreeNode* pOldRight= m_pRight;

    m_pRight= m_pRight->remove(data, bDeleteRight, bRemoveOK);
    if(bDeleteRight)
    {
      delete pOldRight;
    }

    // Data item is removed from the right subtree. The tree
    // root is still this node.
```

```
        pNewRoot= this;
      }

      return pNewRoot;
    }
```

When we remove data from a binary search tree represented by a full tree node, we first search for the data in the tree. If the data is found we delete the node containing the data and re-link the tree properly. However if it is "*this*" node which contains the data to be removed, we cannot delete "*this*" node right away as we are still in its member function. To solve the problem, we have to use a flag *bDeleteThis* to indicate whether "this" node should be deleted or not. When *bDeleteThis* is TRUE, after the function is returned and the node is not needed anymore, we should delete the node. Another flag *bRemoveOK* is used to indicate whether the data is removed successfully. Like function *insert()*, function *remove()* also returns a pointer to LBaseBTreeNode which represents the new root of the binary search tree after the deletion.

If the data to be removed is contained in "*this*" node, we should set *bRemoveOK* and *bDeleteThis* to be TRUE. But what will be the new root of the binary search tree represented by the node? There are several cases: (1) if the left subtree is empty, then we should return *m_pRight* as the new root even if *m_pRight* is a null node; (2) otherwise, if the right subtree is empty, then we should return *m_pLeft* as the new root; (3) if none of the two subtrees are empty, we will move the smallest node in the right subtree to the root position and return the node as the new root. As we know, data in the root node is always greater than all the data in its left subtree and smaller than all the data in its right subtree. When we choose a new root, the data in the new root must satisfy the above condition.

Since the smallest node in the right subtree satisfies this condition and causes minimum re-structuring of the binary search tree, we choose this node as the new root.

If the data to be removed is lesser than the data contained in "*this*" node, we search for the data in the left subtree. Therefore we call *remove()* function recursively through *m_pLeft*. If *m_pLeft* points to a LFullBTreeNode, then the function *LFullBTreeNode::remove()* is called recursively. If *m_pLeft* points to the static null node, then the function *LNullBTreeNode::remove()* is called which will set *bRemoveOK* to be FALSE to indicate that we cannot find the data to be removed from the binary search tree.

If the data to be removed is greater than the data contained in "*this*" node, we search for the data in the right subtree. It works the same way as the above case.

3.1.1.3 Implementing LBTree

Although LBTreeNode already implements the basic operations to operate a binary search tree, it is not straightforward to use. For example, when the user wants to remove a node from a binary search tree, he/she just wants to know whether the node can be successfully removed. He/She is not interested in the pointer returned by function *LBaseBTreeNode::remove()* and the *bDeleteThis* flag. Therefore LBTree is defined to wrap around a LBaseBTreeNode and provide an easy interface for the user to use. Here is the definition of LBTree:

```
class LBTree
{
```

```
// ------------------------------------------------------
// Constructors, destructors and enum types.
// ------------------------------------------------------
public:
  LBTree (void);
  LBTree (const LBTree& rTree) {*this= rTree;}
  virtual ~LBTree(void);

// ------------------------
// Member functions
// ------------------------
public:
  BOOL  remove (const int data);
  void  empty(void);

// operators
  LBTree& operator= (const LBTree& rTree);

// inline functions
  void  insert   (const int data) {m_pRoot= m_pRoot->insert(data);}
  void  traverse  (void) const {m_pRoot->traverse();}
  void  traverse  (ofstream& ofs) const {m_pRoot->traverse(ofs);}
  BOOL  search   (const int data) const {return m_pRoot->search(data);}
  long  space    (void)  { return (sizeof(*this) + m_pRoot->space());}

  const LBaseBTreeNode& getRoot(void) const {return *m_pRoot;}

protected:
  virtual void initRoot(void);
  void setRoot(LBaseBTreeNode* pRoot) { m_pRoot= pRoot;}

// ----------------------
// Attributes
// ----------------------
protected:
  LBaseBTreeNode* m_pRoot;
};
```

A LBTree object has only one member variable *m_pRoot* which is a pointer to LBaseBTreeNode. *m_pRoot* is first initialized to point to the static null node *LFullBTreeNode::m_nullNode_S*. After the first call to *LBTree::insert()*, *m_pRoot* will point to a LFullBTreeNode object.

3.1.2 NULL Pointer Method

In this method, a null tree node is represented by a NULL pointer, we don't need to define a null node class. Two classes are defined:

- LPtrBTreeNode. It represents a full tree node. It contains the node data, and the left and the right links to the subtrees. Initially both the left and the right links are NULL pointers. As we can treat a LPtrBTreeNode object as a valid binary search tree, we mainly implement the binary search tree operations inside this class.
- LPtrBTree. It is a wrapper around a LPtrBTreeNode. The purpose of the class is to provide an easy interface for the user to use.

The implementation of LPtrBTreeNode and LPtrBTree is similar to LFullBTreeNode and LBTree. Both implementing the same set of tree operations and both using recursive function calls. But LPtrBTreeNode cannot take advantage of dynamic binding as LFullBTreeNode does. For example, when implementing function *insert()*, LFullBTreeNode can treat a null tree node and a full tree node the same way -- a tree node.

```
LBaseBTreeNode* LFullBTreeNode::insert
  (const int data)  // new data item to be inserted
{
  if (data < m_data)
  {
    m_pLeft= m_pLeft->insert(data); // recursive function call
  }
  else if (data > m_data)
  {
    m_pRight= m_pRight->insert(data); // recursive function call
  }
return this;
}
```

When we want to insert the data into the left subtree or the right subtree, we simply call *m_pLeft->insert(data)* or *m_pRight->insert(data)*, we do not have to worry that *m_pLeft*

or *m_pRight* points to a null node or a full node. But in LPtrBTreeNode, we must handle the null node case separately.

```
LPtrBTreeNode* LPtrBTreeNode::insert
  (const int data)
{
   if (data < m_data)
   {
     if (m_pLeft == NULL)
     {
       m_pLeft= new LPtrBTreeNode(data);
     }
     else
     {
       m_pLeft->insert(data);
     }
   }
   else if (data > m_data)
   {
     if (m_pRight == NULL)
     {
       m_pRight= new LPtrBTreeNode(data);
     }
     else
     {
       m_pRight->insert(data);
     }
   }
   return this;
}
```

If *m_pLeft* or *m_pRight* is a NULL pointer, we need to create a full tree node, otherwise we can call *LPtrBTreeNode::insert()* recursively.

For more implementation details of LPtrBTreeNode and LPtrBTree, please refer to *LPtrBTreeNode.h* in page 69, *LPtrBTreeNode.cpp* in page 71, *LPtrBTree.h* in page 78, and *LPtrBTree.cpp* in page 80.

3.2 Implementing a Template Binary Search Tree

A binary search tree is a general data structure that can be used to maintain different kinds of data. We can have a binary search tree of integers, doubles, strings, or any other data types. In Section 3.1 (page 11), for simplification, we implement the binary search tree as a binary search tree of integers. In this section, we will generalize the implementation to work with any data types. We will implement the binary search tree using template classes.

3.2.1 Null Node Method

Four template classes are implemented by converting from the implementation of a binary search tree of integers:

TEMPLATE CLASS NAME	CONVERTED FROM
LTmplBaseBTreeNode	LbaseBTreeNode
LTmplNullBTreeNode	LnullBTreeNode
LTmplFullBTreeNode	LfullBTreeNode
LTmplBTree	LBTree

Each template class takes a template parameter DATA_TYPE which is the data type to be maintained by the binary search tree.

The implementation of the template classes is basically the same as the implementation of a binary search tree of integers, but using the template binary search tree LTmplBTree requires that:

- The static member variable *LTmplNullBTreeNode<DATA_TYPE> m_nullNode_S* must be initialized for every DATA_TYPE used in the application. For example, if we have a class called *MyData* and we want to build a binary search tree of *MyData* *LTmplBTree<MyData>*, we must initialize the following

  LTmplNullBTreeNode<MyData> LTmplFullBTreeNode<MyData>::m_nullNode_S;

  The initialization can be done in the *.cpp* file that *LTmplBTree<MyData>* will be used, but it should be done only once in the application.

- The template parameter DATA_TYPE must implement the following operators and constructors.

equality operator ==
inequality operator !=
greater than operator >
greater than or equal operator >=
less than operator <
less than or equal operator <=
output operator <<
assignment operator =
copy constructor if the default one is not good enough.

For implementation details, please refer to *LTmplBTreeNode.h* in page 84, and *LTmplBTree.h* in page 94.

3.2.2 NULL Pointer Method

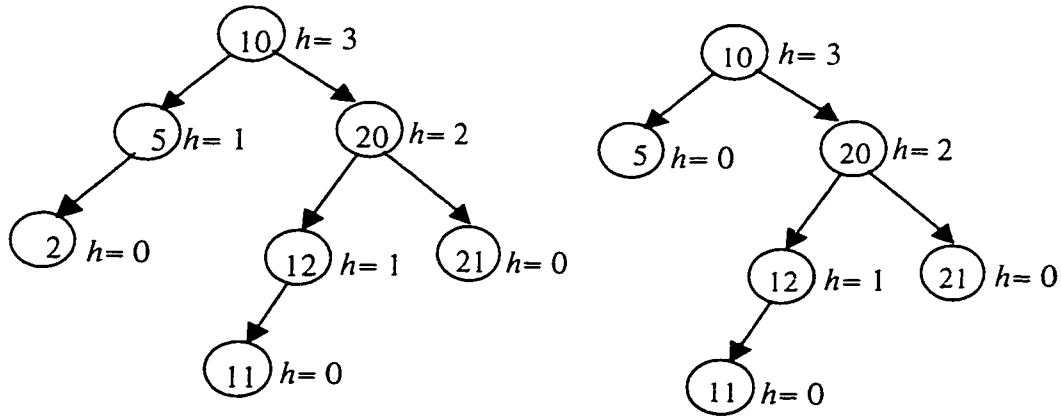
Two template classes are implemented: LTmplPtrBTreeNode and LTmplPtrBTree. LTmplPtrBTreeNode is converted from LPtrBTreeNode, and LTmplPtrBTree is converted from LPtrBTree. Usage of LTmplPtrBTree<DATA_TYPE> is similar to

LTmplBTree<DATA_TYPE>, please refer to Section 3.2.1 in page 23. For implementation details, please refer to *LTmplPtrBTreeNode.h* in page 97 and *LTmplPtrBTree.h* in page 105.

3.3 Implementing an AVL Tree

The formal definition of an AVL tree is as follows: **Height** of a tree is the length of the longest path from the root to a leaf. A binary search tree T is a height-balanced *k*-tree or **HB[*k*]-tree** if each node in the tree has the HB[*k*] property. A node has the **HB[*k*] property** if the height of the left and right subtrees of the node differ in height by at most *k*. A HB[1] tree is also called an **AVL** tree. The following figure shows a valid and an invalid AVL tree.

Figure 3: An AVL Tree Example



(a) Valid AVL Tree **(b) Invalid AVL Tree**

In Figure 3(a), the height difference of the left subtree and the right subtree of any node is no greater than 1, therefore it is a valid AVL tree. In Figure 3(b), the height difference of the left subtree and the right subtree of node 10 is 2, therefore it is not a valid AVL tree.

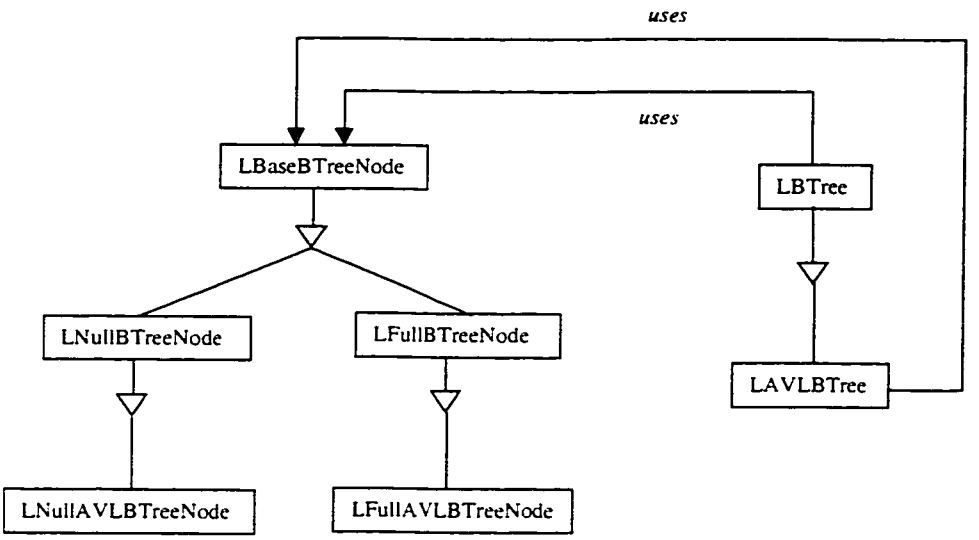
When inserting a node into an AVL tree, we insert it as in an ordinary binary search tree. If the tree becomes unbalanced, we re-balance the tree. This is done by tree rotation. According to different tree patterns, we can apply a left-right single rotation, a left-right double rotation, a right-left single rotation, or a right-left double rotation [8].

When deleting a node from an AVL tree, we initially proceed as discussed in Section 3.1 (page 11). If none of the subtrees of the node to be deleted is empty, we will replace the node to be deleted by the smallest node of its right subtree. After the deletion, any node in the path from the root to the smallest node of the deleted node's right subtree may become unbalanced. Let us denote the path as "*deletion path*". For every node in the deletion path, from bottom to up, we need to check if it is unbalanced. If it is, we need to re-balance the subtree starts from the node. The check will stop when we reach a node whose height is not changed by the deletion. Re-balancing after deletion is also done by rotations as in the insertion case. Actually, we consider that deleting a node from the left/right subtree of a node is equivalent to inserting a node to the right/left subtree of the same node.

3.3.1 Null Node Method

Since an AVL tree is a special binary search tree, we implement an AVL tree by subclassing from the binary search tree classes. The following object diagram shows the relationship of the classes:

Figure 4: Object Diagram of Implementing an AVL Tree



In order to ensure the AVL tree is always balanced, we store the height information at each tree node[2]. First we subclass LNullAVLBTreeNode from LNullBTreeNode.

```
class LNullAVLBTreeNode: public LNullBTreeNode
{
```

---

[2] Some implementations only store the height difference of the left subtree and the right subtree at each tree node.

```
public:
  virtual LBaseBTreeNode* insert(const int data) ;
  virtual int getHeight(void) const {return -1;}
  virtual long space (void) const { return sizeof(*this);}
};
```

A LNullAVLBTreeNode object always has the height of –1. When inserting data into a LNullAVLBTreeNode, function *insert()* returns a pointer to a newly created LFullAVLBTreeNode.

Second we subclass LFullAVLBTreeNode from LFullBTreeNode.

```
class LFullAVLBTreeNode: public LFullBTreeNode
{
// ----------------------------------------
// Constructors, destructors and enum types.
// ----------------------------------------
public:
  LFullAVLBTreeNode(const int newData);
  LFullAVLBTreeNode(const LFullAVLBTreeNode& rNode) : LFullBTreeNode(rNode.getSelfData())
{*this= rNode;}

// ---------
// Methods
// ---------
public:
  virtual LBaseBTreeNode* insert(const int data) ;
  virtual LBaseBTreeNode* remove(const int data, BOOL& bDeleteThis, BOOL& bRemoveOK);

// operators
  LFullAVLBTreeNode& operator= (const LFullAVLBTreeNode& rNode);

  // inlines
  virtual LNullBTreeNode*  getStaticNullNode(void) {return &m_nullAVLNode_S;}
  static  LNullBTreeNode*  getNullNode_S (void) {return &m_nullAVLNode_S;}

  virtual int    getHeight(void) const      {return m_height;}
  virtual void   setHeight(const int height) { m_height= height;}
  virtual long   space (void) const;

public:
  void                  calculateHeight   (void);
  LFullAVLBTreeNode*    balance           (const char direction);

private:
  LFullAVLBTreeNode* detachSmallestNode  (void);
  LFullAVLBTreeNode* singleRotateLeft    (void);
  LFullAVLBTreeNode* doubleRotateLeft    (void);
  LFullAVLBTreeNode* singleRotateRight   (void);
  LFullAVLBTreeNode* doubleRotateRight   (void);
```

```
// ------------
// Attributes
// ------------
private:
  int                          m_height;
  static LNullAVLBTreeNode     m_nullAVLNode_S; // used to represent the leaf. It is static since all the
                                                // leaves can use the same null node.
  static BOOL                  m_bCountNullAVLNodeSpace_S;
};
```

A LFullAVLBTreeNode has a member variable to store the height of the node. It defines a new static null node which is of class LNullAVLBTreeNode. All the empty nodes in the AVL tree will be represented by this new static null node. Class LFullAVLBTreeNode implements functions *singleRotateLeft()*, *doubleRotateLeft()*, *singleRotateRight()*, *doubleRotateRight()* and *balance()* to rotate and balance the tree after insertion and deletion. It also overides *insert()* and *remove()* functions to check the balance property after the insertion/deletion, and re-balance the tree if necessary. It needs to define the assignment operator = too since the operator is not inherited.

Besides the functions that are overridden, LFullAVLBTreeNode should be able to reuse all the functions defined in LFullBTreeNode, such as *search()*, *traverse()*, *removeSubtrees()* etc. However, there is one problem here: the implementation of LFullBTreeNode requires a LNullBTreeNode to represent a null node, but the implementation of LFullAVLBTreeNode requires a LNullAVLBTreeNode to represent a null node. For example, in function *removeSubtrees()*, after deleting all the subtrees, the function will reset the left and the right links to point to the static null node. If the tree node is a LFullBTreeNode, then the links should point to the static LNullBTreeNode. If the tree node is a LFullAVLBTreeNode, then the links should point to the static LNullAVLBTreeNode. To solve this problem, we define a virtual function

getStaticNullNode() in classes LFullBTreeNode and LFullAVLBTreeNode. *LFullBTreeNode::getStaticNullNode()* returns the static LNullBTreeNode. *LFullAVLBTreeNode::getStaticNullNode()* returns the static LNullAVLBTreeNode. Function *removeSubtrees()* calls *getStaticNullNode()* to get the static node to be pointed by the subtree links. Since *getStaticNullNode()* returns a static member variable, it should be a static member function. However C++ does not allow a static member function to be virtual, we have to define the function as a non-static function in this project.

At last, we subclass LAVLBTree from LBTree. LBTree initializes the root to point to the static LNullBTreeNode, but LAVLBTree initializes the root to point to the static LNullAVLBTreeNode.

In order to keep track of the nodes in the deletion path, we implement a template stack class using the null node method. Please refer to *LTmplStack.h* in page 119.

For details of implementing an AVL tree using the null node method, please refer to *LAVLBTreeNode.h* in page 109, *LAVLBTreeNode.cpp* in page 111, *LBTree.h* in page 64, and *LBTree.cpp* in page 66.

3.3.2 NULL Pointer Method

We subclass LPtrAVLBTreeNode from LPtrBTreeNode, and LPtrBTree from LBTree. LPtrAVLBTreeNode stores the height information of the node. It overides functions *insert()* and *remove()*. It implements functions *singleRotateLeft()*, *doubleRotateLeft()*,

singleRotateRight(), *doubleRotateRight()* and *balance()* to re-balance the tree after insertion/deletion. LPtrBTree overrides function *insert()* and creates a new LPtrAVLBTreeNode instead of a LPtrBTreeNode during the first insertion. For implementation details, please refer to *LPtrAVLBTreeNode.h* in page 122, *LPtrAVLBTreeNode.cpp* in page 123, *LPtrBTree.h* in page 78, and *LPtrBTree.cpp* in page 80.

In order to keep track of the nodes in the deletion path, we implement a template stack class using the NULL pointer method. Please refer to *LTmplPtrStack.h* in page 131.

3.4 Implementing a Singly Linked List

A singly linked list is mainly used to maintain data in an unsorted order. There are two ways to add data into a singly linked list: insert data in front of the list or append data at the end of the list.

We use recursive function calls in implementing binary search trees, however we should avoid this in implementing singly linked lists. When we insert to, remove from, search, and traverse a binary search tree, the depth of the recursive function calls is about log(n), where n is the node number. But when we append to, remove from, search and traverse a linked list, the depth of the recursive function calls is n. When n is a large number, recursive function calls will easily cause a stack overflow in the linked list case.

When we build a singly linked list, if we use the *append* method which adds a new node at the end of the list, then the implementation will be extremely inefficient. This is because every time we add a new node, we have to traverse the entire list. We could keep an extra pointer to remember the end of the linked list, but as we implement list operations at the node level, this is not practical. An extra pointer at each node in the list will greatly increase the space requirement. However, if we implement the list operations at the list level, we will not be able to take advantage of null node anymore.

In this project, we use function *insert()* to build large singly linked lists.

3.4.1 Null Node Method

Four classes are implemented: LBaseSListNode, LNullSListNode, LFullSListNode and LSList. Classes LNullSListNode and LFullSListNode are subclassed from LBaseSListNode. LNullSListNode represents a null list node. LFullSListNode represents a full list node. We implement the list operations in LNullSListNode and LFullSListNode. LSList is a wrapper around a LBaseSListNode and is provided for an easy interface for the user to use. For implementation details, please refer to *LSListNode.h* in page 134, *LSListNode.cpp* in page 137, *LSList.h* in page 143, and *LSList.cpp* in page 144.

3.4.2 NULL Pointer Method

Two classes are implemented: LPtrSListNode and LPtrSList. We use a NULL pointer to represent a null list node. LPtrSListNode represents a full list node. We implement the list operations in LPtrSListNode. LPtrSList is a wrapper around a LPtrSListNode and is provided for an easy interface for the user to use. For implementation details, please refer to *LPtrSListNode.h* in page 146, *LPtrSListNode.cpp* in page 148, *LPtrSList.h* in page 153, and *LPtrSList.cpp* in page 154.

4 Experimental Design

In this project, our major interests are comparing the time and space efficiency of the null node method and the NULL pointer method. We also need to make sure that the implementation of the data structures is correct. Therefore designing the experiments includes two steps: (1) Test correctness of the implementation; (2) Compare the time and the space efficiency of the two methods.

4.1 Test Correctness

In *main.cpp* (page 164), a template function *generalTest()* is defined. It is used to test all the data structures implemented in this project, which include:

- LBTree: A binary search tree of integers implemented with the null node method.
- LPtrBTree: A binary search tree of integers implemented with the NULL pointer method.
- LTmplBTree: A template binary search tree implemented with the null node method.
- LTmplPtrBTree: A template binary search tree implemented with the NULL pointer method.
- LAVLBTree: An AVL tree of integers implemented with the null node method.
- LPtrAVLBTee: An AVL tree of integers implemented with the NULL pointer method.
- LSList: A singly linked list of integers implemented with the null node method.

- LPtrSList: A singly linked list of integers implemented with the NULL pointer method.

The following functionality is tested in the template function *generalTest()*:

- Test *insertion*. Starting with an empty data structure, insert ten random numbers into the data structure. Output the data structure and check whether the numbers are correctly inserted.
- Test *assignment operator* =. Use assignment operator = to make a copy of the data structure with ten random numbers and see whether it is correctly copied.
- Test *search*. Insert another five pre-selected numbers into the data structure. Check whether the numbers are correctly inserted. Search for the five pre-selected numbers. Generate a random number, and search for it.
- Test *deletion*. Delete the five pre-selected numbers and check whether they can be correctly deleted.
- Test *traverse*. Each time we do an operation that will change the data structure, we will use the traverse functionality to output the data structure to examine it.
- Test *empty*. Empty the entire data structure and check whether the data structure becomes empty.

Besides function *generalTest()*, another two template functions *slistTest()*, *AVLBTreeTest()* are also implemented in *main.cpp* (page 164) .

In function *slistTest()*, we further test the insertion, appending, and deletion functionality of a singly linked list. We first insert numbers 1 to 10 to the list, then append numbers 1

to 10 to the list. Check the list to see whether the numbers are correctly inserted and appended. We then try to remove 1, 2 and 11. We check whether the numbers are successfully removed and whether the number of nodes removed is correct. At last, we insert five 2s and remove all of them.

In function *AVLBTreeTest()*, we further test single left rotation, single right rotation, double left rotation, double right rotation and deletion of an AVL tree. First we insert numbers 1 to 100, in this order single right rotations will be used to re-balance the tree. Second we generate 20 random numbers within the range of 1 and 100 and remove them from the tree. We check whether the numbers can be successfully removed and the tree is properly re-balanced. We repeat the removing process for 10 times. After testing deletion, we insert numbers 30 to 1 to the tree, in this order single left rotations will be used to re-balance the tree. At last, we empty the tree and rebuild the tree with a special group of numbers. The group of numbers will cause both double left rotations and double right rotations.

The test results are written to the following files:

- *TestTree.txt*. Test result of LBTree. See page 179.
- *TestPtrTree.txt*. Test result of LPtrBTree. See page 182.
- *TestIntTmplTree.txt*. Test result of LTmplBTree<int>. See page 185.
- *TestIntTmplPtrTree.txt*. Test result of LTmplPtrBTree<int>. See page 188.
- *TestAVLTree.txt*. Test result of LAVLBTree. See page 191.
- *TestPtrAVLTree.txt*. Test result of LPtrAVLBTree. See page 202.

- *TestSList.txt*. Test result of LSList. See page 213.
- *TestPtrSList.txt*. Test result of LPtrSList. See page 216.

4.2 Comparing Time and Space Efficiency

Using the null node method eliminates the danger of accessing a NULL pointer, but inheritance and dynamic binding cause certain overhead. We would like to know the time and space performance of the null node method when building and accessing large trees and lists. We also want to compare the results to the equivalent NULL pointer method. We use the following loop to generate some large number. This large number is used as the number of nodes for building trees and lists and the number of times to access trees and lists.

```
for (int j= largeStartNode_G; j <= largeEndNode_G;
    ((j >= 300000) ? (j+= 100000) : (j+= increment_G)))
```

When testing binary trees and AVL trees, we set

```
LargeStartNode_G= 10000;
LargeEndNode_G= 400000;
Increment_G= 20000;
```

When testing linked lists, we set

```
LargeStartNode_G= 10000;
LargeEndNode_G= 80000;
Increment_G= 10000;
```

The numbers chosen for linked lists are smaller than the ones for binary search trees since accessing a linked list is much slower than accessing a binary search tree. Average time of searching a binary search tree is about $\log(n)$ while average time of searching an unsorted singly linked list is about $n/2$.

4.2.1 Calculating Time

We use function *_ftime()*, which gets the current time of the system, to measure the time for building and accessing trees and lists.

To get the build time of a tree or a list, we call *_ftime()* right before and right after building the tree or the list. Then we calculate the time difference between the two calls of *_ftime()*. Similarly, we get the access time of a tree or a list by calling *_ftime()* right before and right after accessing the tree or the list for n times.

4.2.2 Calculating Space

We use *sizeof(*this)* to measure the space needed for each tree node or list node. We implement a *space()* function in each full node class and calculate the space needed for an entire tree or list.

In the null node method, the null node is declared as a static member variable and is allocated only once in the application. Therefore when we calculate the space of a tree or a list, we only count the null node space once in the application.

4.2.3 Comparing Time and Space Efficiency

Given a number n, we define

$$\text{BuildTimeRatio}(n) = \frac{\text{Time to build a data structure with } n \text{ nodes using the NULL pointer method}}{\text{Time to build a data structure with } n \text{ nodes using the null node method}}$$

$$\text{AccessTimeRatio}(n) = \frac{\text{Time to access a data structure with } n \text{ nodes for } n \text{ times using the NULL pointer method}}{\text{Time to access a data structure with } n \text{ nodes for } n \text{ times using the null node method}}$$

$$\text{SpaceRatio}(n) = \frac{\text{Space needed to build a data structure with } n \text{ nodes using the NULL pointer method}}{\text{Space needed to build a data structure with } n \text{ nodes using the null node method}}$$

We use *BuildTimeRatio(n)*, *AccessTimeRatio(n)*, and *SpaceRatio(n)* to compare the time and space efficiency of the null node method and the NULL pointer method. If the ratio is less than 1, it means that the NULL pointer method requires less time or space than the null node method. Since searching in a large linked list may take hours or even days, I modify the definition of *AcessTimeRatio(n)* for a singly linked list as follows:

$$\text{AccessTimeRatio}(n) = \frac{\text{Time to access a data structure with } n \text{ nodes for 500 times using the NULL pointer method}}{\text{Time to access a data structure with } n \text{ nodes for 500 times using the null node method}}$$

The test of the time and the space efficiency is also done in the template function *generalTest()*. For details, please refer to *main.cpp* in page 164.

We defined classes LCompare and LStatistic which are used to keep the time and the space values of each test and calculate the ratios. For details, please refer to *LStatistic.h* in page 157 and *LStatistic.cpp* in page 160.

5 Experimental Results

We did the test three times. Each time a group of test results corresponding to different node numbers is generated. Please refer to Figure 5 (page 45) to Figure 9 (page 49). In this section, we will analyze the test results.

5.1 Build Time Efficiency

We can summarize the build time ratios in the following table:

Table 1: Comparing Build Time of the Two Methods

	Test No.	*Binary search tree of Integers*	*Template Binary Search Tree <Double>*	*Template Binary Search Tree <Int>*	*AVL Tree of Integers*	*Singly Linked List of Integers*
Minimum BuildTimeRatio	1	*1.08627*	*1.10748*	*1.09383*	*1.15557*	*0.860927*
	2	*1.02244*	*1.09257*	*1.12337*	*1.16591*	*0.75*
	3	*1.11033*	*1.05566*	*1.10121*	*1.15355*	*0.857143*
Maximum BuildTimeRatio	1	*1.16667*	*1.15768*	*1.3*	*1.19528*	*1.29032*
	2	*1.33092*	*1.16509*	*1.31965*	*1.18753*	*1*
	3	*1.342*	*1.39073*	*1.38614*	*1.19621*	*0.990991*
Average BuildTimeRatio	1	*1.13698*	*1.13361*	*1.15615*	*1.18567*	*0.958774*
	2	*1.14441*	*1.13462*	*1.17286*	*1.18173*	*0.926097*
	3	*1.16261*	*1.14556*	*1.16706*	*1.18696*	*0.921826*

From the table, we can see that the results of all the three tests are very close. When building a binary search tree or an AVL tree, the null node method is slightly faster than the NULL pointer method. When building a singly linked list, the null node method is slightly slower then the NULL pointer method. Why are such results generated? We will

look at the implementation details of the binary search tree and the singly linked list. First let us look at the binary search tree implementation.

Null Node Method

```
LBaseBTreeNode* LFullBTreeNode::insert
  (const int data) // new data item to be inserted
{
  if (data < m_data)
  {
    m_pLeft= m_pLeft->insert(data);
          // recursive function call
  }
  else if (data > m_data)
  {
    m_pRight= m_pRight->insert(data);
          // recursive function call
  }
  // else {m_data = data} we don't allow
  // duplication of data

  return this;
}
```

```
void LBTree:: insert
  (const int data)
{
  m_pRoot= m_pRoot->insert(data);
}
```

NULL Pointer Method

```
LPtrBTreeNode* LPtrBTreeNode::insert
  (const int data)
{
  if (data < m_data)
  {
    if (m_pLeft != NULL)
    {
      m_pLeft->insert(data);
    }
    else
    {
      m_pLeft= new LPtrBTreeNode(data);
    }
  }
  else if (data > m_data)
  {
    if (m_pRight != NULL)
    {
      m_pRight->insert(data);
    }
    else
    {
      m_pRight= new LPtrBTreeNode(data);
    }
  }

  return this;
}
```

```
void LPtrBTree::insert
  (const int data)
{
  if (m_pRoot == NULL)
  {
    m_pRoot= new LPtrBTreeNode(data);
  }
  else
  {
    m_pRoot= m_pRoot->insert(data);
  }
}
```

When we build a binary search tree, we will call *insert()* function to insert a new node. Each time we insert a new node, the time overhead for the null node method is:

n dynamic bindings + n insert() function calls

the time overhead for the NULL pointer method is:

n tests of NULL + n insert() function calls -- if *m_pRoot* is not NULL

n tests of NULL + (n-1) insert() function calls -- if *m_pRoot* is NULL

n is the depth of recursive calls to function *insert()*. The test results show that the time overhead caused by the dynamic binding in the null node method is slightly smaller than the test of NULL in the NULL pointer method.

Second let us look at the singly linked list implementation.

Null Node Method

```
LBaseSListNode* LFullSListNode::insert
  (const int data)
{
  return new LFullSListNode(data, this);
}


void LSList::insert (const int data)
{
  m_pHead= m_pHead->insert(data);
}
```

NULL Pointer Method

```
LPtrSListNode* LPtrSListNode::insert
  (const int data)
{
  return new LPtrSListNode(data, this);
}


void LPtrSList::insert
  (const int data)
{
  if (m_pHead == NULL)
    m_pHead= new LPtrSListNode(data);
  else
    m_pHead= m_pHead->insert(data);
}
```

When we build a singly linked list, each time we insert a new node, the time overhead for the null node method is:

one dynamic binding + one insert() function call

the time overhead for the NULL pointer method is:

one test of NULL + one insert() function call -- if *m_pHead* is not NULL

one test of NULL -- if *m_pHead* is NULL

From the analysis of the time overhead of the two methods in implementing a singly linked list, we cannot explain why the NULL pointer method performs a little bit better than the null node method. Since the building time of a singly linked list is very short, the *BuildTimeRatio* may not reflect the accurate performance of the two methods. Using the

first test result as an example (please refer to *TestSList.txt* in page 213 and *TestPtrSList.txt* in page 216), the time range of building singly linked lists with nodes from 10000 to 100000 is as follows: the null node method ranges from 31 to 360 milliseconds and the NULL pointer method ranges from 40 to 340 milliseconds.

5.2 Access Time Efficiency

Table 2: Comparing Access Time of the Two Methods

	Test No.	*Binary Search Tree of Integers*	*Template Binary Search Tree <Double>*	*Template Binary Search Tree <Int>*	*AVL Tree of Integers*	*Singly Linked List of Integers*
Minimum AccessTimeRatio	1	*0.974766*	*0.923077*	*0.970588*	*0.99317*	*0.613182*
	2	*0.869146*	*0.923077*	*0.9*	*0.975639*	*0.616844*
	3	*0.968661*	*0.814367*	*0.889488*	*0.992644*	*0.61189*
Maximum AccessTimeRatio	1	*1.09091*	*0.972721*	*1.02102*	*1.02219*	*0.674774*
	2	*1.02145*	*0.970139*	*1.019*	*1.03333*	*0.674535*
	3	*1.11111*	*0.988827*	*1.03244*	*1.0134*	*0.674647*
Average AccessTimeRatio	1	*1.01179*	*0.951768*	*1.00123*	*1.00657*	*0.664287*
	2	*0.979519*	*0.953431*	*0.987465*	*0.996935*	*0.664628*
	3	*1.00618*	*0.943071*	*0.991536*	*1.0024*	*0.664091*

From the test results, the average *AccessTimeRatio* of binary search trees, template binary search trees and AVL trees are almost 1. But the average *AccessTimeRatio* of the singly linked lists is only around 0.66. We can look at the implementation of the singly linked list and see why the null node method is much slower than the NULL pointer method in this case.

Null Node Method

```
BOOL LFullSListNode::search
  (const int data)
{
  if (m_data == data)
    return TRUE;

  LBaseSListNode *pNext= m_pNext;

  while (!pNext->isNull())
  {
    if (data == pNext->getSelfData())
      return TRUE;

    pNext= pNext->getNext();
  }

  return FALSE;

}
```

NULL Pointer Method

```
BOOL LPtrSListNode::search
  (const int data)
{
  if (m_data == data)
    return TRUE;

  LPtrSListNode *pNext= m_pNext;

  while (pNext != NULL)
  {
    if (data == pNext->getSelfData())
      return TRUE;

    pNext= pNext->getNext();

  }

  return FALSE;
}
```

In the null node case, the function calls *isNull()*, *getSelfData()*, *getNext()* all involve dynamic binding, while in the NULL pointer case do not. This explains why the null node method is much slower for linked lists.

5.3 Space Efficiency

From the test results, we can see that the space ratios of all the data structures we implemented are 1 except for node number 10000. This is because the null node method needs to allocate extra space for the static null node, but it is only allocated once in the application. The extra bytes allocated for the static null node is counted when we first calculate the space of a tree or a list; afterwards it is not counted anymore. For large trees and lists, we can ignore the few bytes allocated for the static null node, and consider the space efficiency of the two methods are the same.

Figure 5: Test Result of Binary Search Tree of Integers

First Test	Second Test	Third Test
Build Time Ratio: --- *BuildTimeRatio(10000)= 1.16667* *BuildTimeRatio(30000)= 1.10782* *BuildTimeRatio(50000)= 1.14219* *BuildTimeRatio(70000)= 1.1594* *BuildTimeRatio(90000)= 1.1368* *BuildTimeRatio(110000)= 1.15248* *BuildTimeRatio(130000)= 1.15633* *BuildTimeRatio(150000)= 1.13003* *BuildTimeRatio(170000)= 1.15177* *BuildTimeRatio(190000)= 1.15207* *BuildTimeRatio(210000)= 1.08627* *BuildTimeRatio(230000)= 1.15451* *BuildTimeRatio(250000)= 1.11483* *BuildTimeRatio(270000)= 1.13293* *BuildTimeRatio(290000)= 1.12704* *BuildTimeRatio(310000)= 1.1206* *BuildTimeRatio(min)= 1.08627* *BuildTimeRatio(max)= 1.16667* *BuildTimeRatio(ave)= 1.13698* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 1* *AccessTimeRatio(30000)= 1.09091* *AccessTimeRatio(50000)= 1* *AccessTimeRatio(70000)= 0.989142* *AccessTimeRatio(90000)= 1.02517* *AccessTimeRatio(110000)= 1.00678* *AccessTimeRatio(130000)= 1.02653* *AccessTimeRatio(150000)= 0.978568* *AccessTimeRatio(170000)= 0.974766* *AccessTimeRatio(190000)= 1.00339* *AccessTimeRatio(210000)= 1.01196* *AccessTimeRatio(230000)= 1.02415* *AccessTimeRatio(250000)= 1* *AccessTimeRatio(270000)= 1.0437* *AccessTimeRatio(290000)= 1.02102* *AccessTimeRatio(310000)= 0.992524* *AccessTimeRatio(min)= 0.974766* *AccessTimeRatio(max)= 1.09091* *AccessTimeRatio(ave)= 1.01179* *Space Ratio:* --- *SpaceRatio(10000)= 0.999975* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.999975* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999998*	*Build Time Ratio:* --- *BuildTimeRatio(10000)= 1.16667* *BuildTimeRatio(30000)= 1.02244* *BuildTimeRatio(50000)= 1.1248* *BuildTimeRatio(70000)= 1.13491* *BuildTimeRatio(90000)= 1.33092* *BuildTimeRatio(110000)= 1.17544* *BuildTimeRatio(130000)= 1.1765* *BuildTimeRatio(150000)= 1.15474* *BuildTimeRatio(170000)= 1.18555* *BuildTimeRatio(190000)= 1.13087* *BuildTimeRatio(210000)= 1.11793* *BuildTimeRatio(230000)= 1.11747* *BuildTimeRatio(250000)= 1.098* *BuildTimeRatio(270000)= 1.14219* *BuildTimeRatio(290000)= 1.0969* *BuildTimeRatio(310000)= 1.13524* *BuildTimeRatio(min)= 1.02244* *BuildTimeRatio(max)= 1.33092* *BuildTimeRatio(ave)= 1.14441* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 0.91* *AccessTimeRatio(30000)= 1* *AccessTimeRatio(50000)= 0.984152* *AccessTimeRatio(70000)= 1.00987* *AccessTimeRatio(90000)= 0.869146* *AccessTimeRatio(110000)= 0.953542* *AccessTimeRatio(130000)= 0.914144* *AccessTimeRatio(150000)= 1.02145* *AccessTimeRatio(170000)= 1.01842* *AccessTimeRatio(190000)= 1.00677* *AccessTimeRatio(210000)= 0.994039* *AccessTimeRatio(230000)= 1.00808* *AccessTimeRatio(250000)= 0.99486* *AccessTimeRatio(270000)= 1.00198* *AccessTimeRatio(290000)= 0.989663* *AccessTimeRatio(310000)= 0.996182* *AccessTimeRatio(min)= 0.869146* *AccessTimeRatio(max)= 1.02145* *AccessTimeRatio(ave)= 0.979519* *Space Ratio:* --- *SpaceRatio(10000)= 0.999975* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.999975* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999998*	*Build Time Ratio:* --- *BuildTimeRatio(10000)= 1.17117* *BuildTimeRatio(30000)= 1.19189* *BuildTimeRatio(50000)= 1.13825* *BuildTimeRatio(70000)= 1.16824* *BuildTimeRatio(90000)= 1.342* *BuildTimeRatio(110000)= 1.17218* *BuildTimeRatio(130000)= 1.12838* *BuildTimeRatio(150000)= 1.14188* *BuildTimeRatio(170000)= 1.18759* *BuildTimeRatio(190000)= 1.15906* *BuildTimeRatio(210000)= 1.14585* *BuildTimeRatio(230000)= 1.15381* *BuildTimeRatio(250000)= 1.1195* *BuildTimeRatio(270000)= 1.14587* *BuildTimeRatio(290000)= 1.1258* *BuildTimeRatio(310000)= 1.11033* *BuildTimeRatio(min)= 1.11033* *BuildTimeRatio(max)= 1.342* *BuildTimeRatio(ave)= 1.16261* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 1.11111* *AccessTimeRatio(30000)= 0.968661* *AccessTimeRatio(50000)= 0.983897* *AccessTimeRatio(70000)= 0.998915* *AccessTimeRatio(90000)= 1* *AccessTimeRatio(110000)= 1.01935* *AccessTimeRatio(130000)= 1.00983* *AccessTimeRatio(150000)= 0.982706* *AccessTimeRatio(170000)= 1.00404* *AccessTimeRatio(190000)= 1.01005* *AccessTimeRatio(210000)= 1* *AccessTimeRatio(230000)= 0.972087* *AccessTimeRatio(250000)= 0.997582* *AccessTimeRatio(270000)= 1.02816* *AccessTimeRatio(290000)= 1.01248* *AccessTimeRatio(310000)= 1* *AccessTimeRatio(min)= 0.968661* *AccessTimeRatio(max)= 1.11111* *AccessTimeRatio(ave)= 1.00618* *Space Ratio:* --- *SpaceRatio(10000)= 0.999975* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.999975* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999998*

Figure 6: Test Result of Template Binary Search Tree of Doubles

First Test	Second Test	Third Test
Build Time Ratio: *----------* *BuildTimeRatio(10000)= 1.13333* *BuildTimeRatio(30000)= 1.15768* *BuildTimeRatio(50000)= 1.15215* *BuildTimeRatio(70000)= 1.1491* *BuildTimeRatio(90000)= 1.11229* *BuildTimeRatio(110000)= 1.1511* *BuildTimeRatio(130000)= 1.15326* *BuildTimeRatio(150000)= 1.12678* *BuildTimeRatio(170000)= 1.1367* *BuildTimeRatio(190000)= 1.12444* *BuildTimeRatio(210000)= 1.14193* *BuildTimeRatio(230000)= 1.11938* *BuildTimeRatio(250000)= 1.11275* *BuildTimeRatio(270000)= 1.1355* *BuildTimeRatio(290000)= 1.10748* *BuildTimeRatio(310000)= 1.12388* *BuildTimeRatio(min)= 1.10748* *BuildTimeRatio(max)= 1.15768* *BuildTimeRatio(ave)= 1.13361*	*Build Time Ratio:* *----------* *BuildTimeRatio(10000)= 1.13333* *BuildTimeRatio(30000)= 1.14553* *BuildTimeRatio(50000)= 1.16073* *BuildTimeRatio(70000)= 1.16509* *BuildTimeRatio(90000)= 1.13012* *BuildTimeRatio(110000)= 1.16379* *BuildTimeRatio(130000)= 1.15844* *BuildTimeRatio(150000)= 1.12642* *BuildTimeRatio(170000)= 1.14536* *BuildTimeRatio(190000)= 1.10179* *BuildTimeRatio(210000)= 1.12425* *BuildTimeRatio(230000)= 1.1272* *BuildTimeRatio(250000)= 1.09257* *BuildTimeRatio(270000)= 1.13903* *BuildTimeRatio(290000)= 1.11526* *BuildTimeRatio(310000)= 1.12496* *BuildTimeRatio(min)= 1.09257* *BuildTimeRatio(max)= 1.16509* *BuildTimeRatio(ave)= 1.13462*	*Build Time Ratio:* *----------* *BuildTimeRatio(10000)= 1.39073* *BuildTimeRatio(30000)= 1.05566* *BuildTimeRatio(50000)= 1.07709* *BuildTimeRatio(70000)= 1.32437* *BuildTimeRatio(90000)= 1.14176* *BuildTimeRatio(110000)= 1.14008* *BuildTimeRatio(130000)= 1.14407* *BuildTimeRatio(150000)= 1.13622* *BuildTimeRatio(170000)= 1.12189* *BuildTimeRatio(190000)= 1.12476* *BuildTimeRatio(210000)= 1.13155* *BuildTimeRatio(230000)= 1.11721* *BuildTimeRatio(250000)= 1.11014* *BuildTimeRatio(270000)= 1.10355* *BuildTimeRatio(290000)= 1.10164* *BuildTimeRatio(310000)= 1.1082* *BuildTimeRatio(min)= 1.05566* *BuildTimeRatio(max)= 1.39073* *BuildTimeRatio(ave)= 1.14556*
Access Time Ratio: *----------* *AccessTimeRatio(10000)= 0.923077* *AccessTimeRatio(30000)= 0.939583* *AccessTimeRatio(50000)= 0.965157* *AccessTimeRatio(70000)= 0.972721* *AccessTimeRatio(90000)= 0.963258* *AccessTimeRatio(110000)= 0.962669* *AccessTimeRatio(130000)= 0.961214* *AccessTimeRatio(150000)= 0.938684* *AccessTimeRatio(170000)= 0.950762* *AccessTimeRatio(190000)= 0.931396* *AccessTimeRatio(210000)= 0.961373* *AccessTimeRatio(230000)= 0.958059* *AccessTimeRatio(250000)= 0.933489* *AccessTimeRatio(270000)= 0.963504* *AccessTimeRatio(290000)= 0.943565* *AccessTimeRatio(310000)= 0.95977* *AccessTimeRatio(min)= 0.923077* *AccessTimeRatio(max)= 0.972721* *AccessTimeRatio(ave)= 0.951768*	*Access Time Ratio:* *----------* *AccessTimeRatio(10000)= 0.923077* *AccessTimeRatio(30000)= 0.96008* *AccessTimeRatio(50000)= 0.943946* *AccessTimeRatio(70000)= 0.953704* *AccessTimeRatio(90000)= 0.946126* *AccessTimeRatio(110000)= 0.963014* *AccessTimeRatio(130000)= 0.950627* *AccessTimeRatio(150000)= 0.938191* *AccessTimeRatio(170000)= 0.956224* *AccessTimeRatio(190000)= 0.967135* *AccessTimeRatio(210000)= 0.961763* *AccessTimeRatio(230000)= 0.954428* *AccessTimeRatio(250000)= 0.961199* *AccessTimeRatio(270000)= 0.970139* *AccessTimeRatio(290000)= 0.963486* *AccessTimeRatio(310000)= 0.941761* *AccessTimeRatio(min)= 0.923077* *AccessTimeRatio(max)= 0.970139* *AccessTimeRatio(ave)= 0.953431*	*Access Time Ratio:* *----------* *AccessTimeRatio(10000)= 0.864286* *AccessTimeRatio(30000)= 0.960417* *AccessTimeRatio(50000)= 0.965948* *AccessTimeRatio(70000)= 0.814367* *AccessTimeRatio(90000)= 0.959816* *AccessTimeRatio(110000)= 0.977406* *AccessTimeRatio(130000)= 0.973743* *AccessTimeRatio(150000)= 0.955906* *AccessTimeRatio(170000)= 0.938065* *AccessTimeRatio(190000)= 0.93634* *AccessTimeRatio(210000)= 0.963388* *AccessTimeRatio(230000)= 0.934543* *AccessTimeRatio(250000)= 0.960701* *AccessTimeRatio(270000)= 0.939526* *AccessTimeRatio(290000)= 0.955853* *AccessTimeRatio(310000)= 0.988827* *AccessTimeRatio(min)= 0.814367* *AccessTimeRatio(max)= 0.988827* *AccessTimeRatio(ave)= 0.943071*
Space Ratio: *----------* *SpaceRatio(10000)= 0.999983* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.999983* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999999*	*Space Ratio:* *----------* *SpaceRatio(10000)= 0.999983* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.999983* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999999*	*Space Ratio:* *----------* *SpaceRatio(10000)= 0.999983* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.999983* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999999*

Figure 7: Test Result of Template Binary Search Tree of Integers

First Test	Second Test	Third Test
Build Time Ratio: --- *BuildTimeRatio(10000)= 1.3* *BuildTimeRatio(30000)= 1.16173* *BuildTimeRatio(50000)= 1.19324* *BuildTimeRatio(70000)= 1.18066* *BuildTimeRatio(90000)= 1.09383* *BuildTimeRatio(110000)= 1.15922* *BuildTimeRatio(130000)= 1.13841* *BuildTimeRatio(150000)= 1.16212* *BuildTimeRatio(170000)= 1.16739* *BuildTimeRatio(190000)= 1.14075* *BuildTimeRatio(210000)= 1.13062* *BuildTimeRatio(230000)= 1.14625* *BuildTimeRatio(250000)= 1.12276* *BuildTimeRatio(270000)= 1.13356* *BuildTimeRatio(290000)= 1.14431* *BuildTimeRatio(310000)= 1.12362* *BuildTimeRatio(min)= 1.09383* *BuildTimeRatio(max)= 1.3* *BuildTimeRatio(ave)= 1.15615* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 1* *AccessTimeRatio(30000)= 0.970588* *AccessTimeRatio(50000)= 1.00164* *AccessTimeRatio(70000)= 1* *AccessTimeRatio(90000)= 0.991681* *AccessTimeRatio(110000)= 1.01896* *AccessTimeRatio(130000)= 0.995342* *AccessTimeRatio(150000)= 0.986917* *AccessTimeRatio(170000)= 1.01076* *AccessTimeRatio(190000)= 1.00034* *AccessTimeRatio(210000)= 0.997001* *AccessTimeRatio(230000)= 1.01074* *AccessTimeRatio(250000)= 1* *AccessTimeRatio(270000)= 1.00682* *AccessTimeRatio(290000)= 1.02102* *AccessTimeRatio(310000)= 1.00784* *AccessTimeRatio(min)= 0.970588* *AccessTimeRatio(max)= 1.02102* *AccessTimeRatio(ave)= 1.00123* *Space Ratio:* --- *SpaceRatio(10000)= 0.999975* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.999975* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999998*	*Build Time Ratio:* --- *BuildTimeRatio(10000)= 1.28182* *BuildTimeRatio(30000)= 1.31965* *BuildTimeRatio(50000)= 1.20934* *BuildTimeRatio(70000)= 1.18172* *BuildTimeRatio(90000)= 1.13655* *BuildTimeRatio(110000)= 1.1602* *BuildTimeRatio(130000)= 1.16641* *BuildTimeRatio(150000)= 1.14401* *BuildTimeRatio(170000)= 1.1634* *BuildTimeRatio(190000)= 1.13748* *BuildTimeRatio(210000)= 1.14714* *BuildTimeRatio(230000)= 1.15682* *BuildTimeRatio(250000)= 1.13402* *BuildTimeRatio(270000)= 1.1502* *BuildTimeRatio(290000)= 1.15367* *BuildTimeRatio(310000)= 1.12337* *BuildTimeRatio(min)= 1.12337* *BuildTimeRatio(max)= 1.31965* *BuildTimeRatio(ave)= 1.17286* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 0.9* *AccessTimeRatio(30000)= 0.919444* *AccessTimeRatio(50000)= 0.969841* *AccessTimeRatio(70000)= 1* *AccessTimeRatio(90000)= 0.975248* *AccessTimeRatio(110000)= 1.0062* *AccessTimeRatio(130000)= 0.990266* *AccessTimeRatio(150000)= 0.982992* *AccessTimeRatio(170000)= 1* *AccessTimeRatio(190000)= 0.996638* *AccessTimeRatio(210000)= 1.01517* *AccessTimeRatio(230000)= 1.01098* *AccessTimeRatio(250000)= 1.00246* *AccessTimeRatio(270000)= 1.01328* *AccessTimeRatio(290000)= 1.019* *AccessTimeRatio(310000)= 0.997908* *AccessTimeRatio(min)= 0.9* *AccessTimeRatio(max)= 1.019* *AccessTimeRatio(ave)= 0.987465* *Space Ratio:* --- *SpaceRatio(10000)= 0.999975* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.999975* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999998*	*Build Time Ratio:* --- *BuildTimeRatio(10000)= 1.38614* *BuildTimeRatio(30000)= 1.225* *BuildTimeRatio(50000)= 1.17433* *BuildTimeRatio(70000)= 1.15609* *BuildTimeRatio(90000)= 1.10121* *BuildTimeRatio(110000)= 1.17218* *BuildTimeRatio(130000)= 1.16106* *BuildTimeRatio(150000)= 1.11278* *BuildTimeRatio(170000)= 1.17466* *BuildTimeRatio(190000)= 1.14393* *BuildTimeRatio(210000)= 1.14876* *BuildTimeRatio(230000)= 1.14964* *BuildTimeRatio(250000)= 1.14051* *BuildTimeRatio(270000)= 1.15563* *BuildTimeRatio(290000)= 1.14837* *BuildTimeRatio(310000)= 1.12271* *BuildTimeRatio(min)= 1.10121* *BuildTimeRatio(max)= 1.38614* *BuildTimeRatio(ave)= 1.16706* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 0.9* *AccessTimeRatio(30000)= 0.889488* *AccessTimeRatio(50000)= 1* *AccessTimeRatio(70000)= 0.998915* *AccessTimeRatio(90000)= 0.967532* *AccessTimeRatio(110000)= 1.03244* *AccessTimeRatio(130000)= 1.02548* *AccessTimeRatio(150000)= 0.99562* *AccessTimeRatio(170000)= 1.01147* *AccessTimeRatio(190000)= 1* *AccessTimeRatio(210000)= 1.00626* *AccessTimeRatio(230000)= 1.01351* *AccessTimeRatio(250000)= 1* *AccessTimeRatio(270000)= 1.01305* *AccessTimeRatio(290000)= 1.01251* *AccessTimeRatio(310000)= 0.998294* *AccessTimeRatio(min)= 0.889488* *AccessTimeRatio(max)= 1.03244* *AccessTimeRatio(ave)= 0.991536* *Space Ratio:* --- *SpaceRatio(10000)= 0.999975* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.999975* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999998*

Figure 8: Test Result of AVL Tree of Integers

First Test	Second Test	Third Test
Build Time Ratio: --- *BuildTimeRatio(10000)= 1.19194* *BuildTimeRatio(30000)= 1.15557* *BuildTimeRatio(50000)= 1.16923* *BuildTimeRatio(70000)= 1.18756* *BuildTimeRatio(90000)= 1.19528* *BuildTimeRatio(110000)= 1.18978* *BuildTimeRatio(130000)= 1.1825* *BuildTimeRatio(150000)= 1.1836* *BuildTimeRatio(170000)= 1.19211* *BuildTimeRatio(190000)= 1.1909* *BuildTimeRatio(210000)= 1.19209* *BuildTimeRatio(230000)= 1.19135* *BuildTimeRatio(250000)= 1.18644* *BuildTimeRatio(270000)= 1.18952* *BuildTimeRatio(290000)= 1.18463* *BuildTimeRatio(310000)= 1.18826* *BuildTimeRatio(min)= 1.15557* *BuildTimeRatio(max)= 1.19528* *BuildTimeRatio(ave)= 1.18567* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 1* *AccessTimeRatio(30000)= 0.996678* *AccessTimeRatio(50000)= 1.00182* *AccessTimeRatio(70000)= 1.01248* *AccessTimeRatio(90000)= 1.01848* *AccessTimeRatio(110000)= 1.02219* *AccessTimeRatio(130000)= 1* *AccessTimeRatio(150000)= 1.00463* *AccessTimeRatio(170000)= 1.00446* *AccessTimeRatio(190000)= 1.0078* *AccessTimeRatio(210000)= 1.00346* *AccessTimeRatio(230000)= 1.01228* *AccessTimeRatio(250000)= 1.00574* *AccessTimeRatio(270000)= 1.01054* *AccessTimeRatio(290000)= 0.99317* *AccessTimeRatio(310000)= 1.01132* *AccessTimeRatio(min)= 0.99317* *AccessTimeRatio(max)= 1.02219* *AccessTimeRatio(ave)= 1.00657* *Space Ratio:* --- *SpaceRatio(10000)= 0.99998* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.99998* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999999*	*Build Time Ratio:* --- *BuildTimeRatio(10000)= 1.17964* *BuildTimeRatio(30000)= 1.18465* *BuildTimeRatio(50000)= 1.16591* *BuildTimeRatio(70000)= 1.1842* *BuildTimeRatio(90000)= 1.18515* *BuildTimeRatio(110000)= 1.18095* *BuildTimeRatio(130000)= 1.17677* *BuildTimeRatio(150000)= 1.18307* *BuildTimeRatio(170000)= 1.18722* *BuildTimeRatio(190000)= 1.18753* *BuildTimeRatio(210000)= 1.18467* *BuildTimeRatio(230000)= 1.1807* *BuildTimeRatio(250000)= 1.18087* *BuildTimeRatio(270000)= 1.18347* *BuildTimeRatio(290000)= 1.18002* *BuildTimeRatio(310000)= 1.18285* *BuildTimeRatio(min)= 1.16591* *BuildTimeRatio(max)= 1.18753* *BuildTimeRatio(ave)= 1.18173* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 1* *AccessTimeRatio(30000)= 1.03333* *AccessTimeRatio(50000)= 0.981851* *AccessTimeRatio(70000)= 0.975639* *AccessTimeRatio(90000)= 0.980944* *AccessTimeRatio(110000)= 1* *AccessTimeRatio(130000)= 0.987966* *AccessTimeRatio(150000)= 1.00515* *AccessTimeRatio(170000)= 1.00444* *AccessTimeRatio(190000)= 0.988003* *AccessTimeRatio(210000)= 1* *AccessTimeRatio(230000)= 1.00977* *AccessTimeRatio(250000)= 0.977496* *AccessTimeRatio(270000)= 0.999738* *AccessTimeRatio(290000)= 0.997594* *AccessTimeRatio(310000)= 1.00904* *AccessTimeRatio(min)= 0.975639* *AccessTimeRatio(max)= 1.03333* *AccessTimeRatio(ave)= 0.996935* *Space Ratio:* --- *SpaceRatio(10000)= 0.99998* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.99998* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999999*	*Build Time Ratio:* --- *BuildTimeRatio(10000)= 1.15355* *BuildTimeRatio(30000)= 1.17379* *BuildTimeRatio(50000)= 1.17882* *BuildTimeRatio(70000)= 1.18713* *BuildTimeRatio(90000)= 1.19304* *BuildTimeRatio(110000)= 1.18541* *BuildTimeRatio(130000)= 1.19312* *BuildTimeRatio(150000)= 1.18788* *BuildTimeRatio(170000)= 1.19583* *BuildTimeRatio(190000)= 1.19339* *BuildTimeRatio(210000)= 1.19132* *BuildTimeRatio(230000)= 1.19145* *BuildTimeRatio(250000)= 1.18925* *BuildTimeRatio(270000)= 1.19621* *BuildTimeRatio(290000)= 1.19081* *BuildTimeRatio(310000)= 1.19036* *BuildTimeRatio(min)= 1.15355* *BuildTimeRatio(max)= 1.19621* *BuildTimeRatio(ave)= 1.18696* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 1* *AccessTimeRatio(30000)= 0.996678* *AccessTimeRatio(50000)= 1* *AccessTimeRatio(70000)= 0.998768* *AccessTimeRatio(90000)= 1.00093* *AccessTimeRatio(110000)= 1* *AccessTimeRatio(130000)= 1* *AccessTimeRatio(150000)= 1* *AccessTimeRatio(170000)= 1.00484* *AccessTimeRatio(190000)= 0.992644* *AccessTimeRatio(210000)= 1.00346* *AccessTimeRatio(230000)= 1.00593* *AccessTimeRatio(250000)= 1.00569* *AccessTimeRatio(270000)= 1.0134* *AccessTimeRatio(290000)= 1.00482* *AccessTimeRatio(310000)= 1.01127* *AccessTimeRatio(min)= 0.992644* *AccessTimeRatio(max)= 1.0134* *AccessTimeRatio(ave)= 1.0024* *Space Ratio:* --- *SpaceRatio(10000)= 0.99998* *SpaceRatio(30000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(110000)= 1* *SpaceRatio(130000)= 1* *SpaceRatio(150000)= 1* *SpaceRatio(170000)= 1* *SpaceRatio(190000)= 1* *SpaceRatio(210000)= 1* *SpaceRatio(230000)= 1* *SpaceRatio(250000)= 1* *SpaceRatio(270000)= 1* *SpaceRatio(290000)= 1* *SpaceRatio(310000)= 1* *SpaceRatio(min)= 0.99998* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999999*

Figure 9: Test Result of Singly Linked List of Integers

First Test	Second Test	Third Test
Build Time Ratio: --- *BuildTimeRatio(10000)= 1.29032* *BuildTimeRatio(20000)= 1* *BuildTimeRatio(30000)= 0.909091* *BuildTimeRatio(40000)= 0.860927* *BuildTimeRatio(50000)= 0.944444* *BuildTimeRatio(60000)= 0.909091* *BuildTimeRatio(70000)= 0.923077* *BuildTimeRatio(80000)= 0.9* *BuildTimeRatio(90000)= 0.906344* *BuildTimeRatio(100000)= 0.944444* *BuildTimeRatio(min)= 0.860927* *BuildTimeRatio(max)= 1.29032* *BuildTimeRatio(ave)= 0.958774* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 0.613182* *AccessTimeRatio(20000)= 0.649566* *AccessTimeRatio(30000)= 0.664664* *AccessTimeRatio(40000)= 0.674774* *AccessTimeRatio(50000)= 0.673359* *AccessTimeRatio(60000)= 0.6737* *AccessTimeRatio(70000)= 0.673414* *AccessTimeRatio(80000)= 0.673663* *AccessTimeRatio(90000)= 0.673889* *AccessTimeRatio(100000)= 0.672664* *AccessTimeRatio(min)= 0.613182* *AccessTimeRatio(max)= 0.674774* *AccessTimeRatio(ave)= 0.664287* *Space Ratio:* --- *SpaceRatio(10000)= 0.999967* *SpaceRatio(20000)= 1* *SpaceRatio(30000)= 1* *SpaceRatio(40000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(60000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(80000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(100000)= 1* *SpaceRatio(min)= 0.999967* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999997*	*Build Time Ratio:* --- *BuildTimeRatio(10000)= 0.75* *BuildTimeRatio(20000)= 1* *BuildTimeRatio(30000)= 1* *BuildTimeRatio(40000)= 1* *BuildTimeRatio(50000)= 0.889503* *BuildTimeRatio(60000)= 0.904977* *BuildTimeRatio(70000)= 0.926923* *BuildTimeRatio(80000)= 0.931034* *BuildTimeRatio(90000)= 0.939394* *BuildTimeRatio(100000)= 0.919137* *BuildTimeRatio(min)= 0.75* *BuildTimeRatio(max)= 1* *BuildTimeRatio(ave)= 0.926097* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 0.616844* *AccessTimeRatio(20000)= 0.649848* *AccessTimeRatio(30000)= 0.664043* *AccessTimeRatio(40000)= 0.674535* *AccessTimeRatio(50000)= 0.673532* *AccessTimeRatio(60000)= 0.673072* *AccessTimeRatio(70000)= 0.674364* *AccessTimeRatio(80000)= 0.673699* *AccessTimeRatio(90000)= 0.673014* *AccessTimeRatio(100000)= 0.673326* *AccessTimeRatio(min)= 0.616844* *AccessTimeRatio(max)= 0.674535* *AccessTimeRatio(ave)= 0.664628* *Space Ratio:* --- *SpaceRatio(10000)= 0.999967* *SpaceRatio(20000)= 1* *SpaceRatio(30000)= 1* *SpaceRatio(40000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(60000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(80000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(100000)= 1* *SpaceRatio(min)= 0.999967* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999997*	*Build Time Ratio:* --- *BuildTimeRatio(10000)= 0.97561* *BuildTimeRatio(20000)= 0.857143* *BuildTimeRatio(30000)= 0.990991* *BuildTimeRatio(40000)= 0.927152* *BuildTimeRatio(50000)= 0.944444* *BuildTimeRatio(60000)= 0.913043* *BuildTimeRatio(70000)= 0.884615* *BuildTimeRatio(80000)= 0.9* *BuildTimeRatio(90000)= 0.906344* *BuildTimeRatio(100000)= 0.918919* *BuildTimeRatio(min)= 0.857143* *BuildTimeRatio(max)= 0.990991* *BuildTimeRatio(ave)= 0.921826* *Access Time Ratio:* --- *AccessTimeRatio(10000)= 0.61189* *AccessTimeRatio(20000)= 0.649112* *AccessTimeRatio(30000)= 0.663026* *AccessTimeRatio(40000)= 0.674295* *AccessTimeRatio(50000)= 0.674345* *AccessTimeRatio(60000)= 0.673209* *AccessTimeRatio(70000)= 0.673834* *AccessTimeRatio(80000)= 0.674647* *AccessTimeRatio(90000)= 0.672901* *AccessTimeRatio(100000)= 0.67365* *AccessTimeRatio(min)= 0.61189* *AccessTimeRatio(max)= 0.674647* *AccessTimeRatio(ave)= 0.664091* *Space Ratio:* --- *SpaceRatio(10000)= 0.999967* *SpaceRatio(20000)= 1* *SpaceRatio(30000)= 1* *SpaceRatio(40000)= 1* *SpaceRatio(50000)= 1* *SpaceRatio(60000)= 1* *SpaceRatio(70000)= 1* *SpaceRatio(80000)= 1* *SpaceRatio(90000)= 1* *SpaceRatio(100000)= 1* *SpaceRatio(min)= 0.999967* *SpaceRatio(max)= 1* *SpaceRatio(ave)= 0.999997*

6 Conclusions

The null node method provides a new way to look at the traditional NULL pointer problem. We apply the Object-Oriented method and use a null node to replace a NULL pointer. Unlike accessing a NULL pointer, accessing a null node is not dangerous.

The null node method requires class inheritance and the use of virtual functions. The implementation is a little bit more complex than the NULL pointer method. For example, if we use the NULL pointer method to implement a binary search tree, we need to write one class to represent a tree node. But if use the null node method, we need to write three classes – the base node class, the null node class and the full node class. Although three classes needed to be written, the implementation of the base node class and the null node class is simple. The base node class only needs to define the node operations as pure virtual functions, and the null node class only needs to give default response or output the error which the NULL pointer method does not provide. Therefore the actual null node implementation is not as complex as it appears at the first look. In another aspect, the null node method can even simplify the code. For example, when we insert a node or search for some data, the null node method does not require the check of NULL anymore, while the NULL pointer method always needs to check for NULL before accessing a pointer.

As shown in Section 5 (page 40), we can see that the dynamic binding does not introduce much overhead to the time performance. Except for accessing a singly linked list, the null node method either performs a little bit better or about the same as the NULL pointer

method. Since a singly linked list only has one null node, it cannot take good advantage of the null node method. We would say binary search trees are better applications to apply the null node method.

When the node number is large, we can ignore the extra space occupied by the static null node. Therefore we can say that the space efficiency of the two methods is the same.

On the whole, by applying the Object-Oriented Design method, the null node method provides a new solution to the NULL pointer problem without introducing much time and space overhead. But the null node method is not a perfect solution to the NULL pointer problem. In order to use the null node method efficiently and conveniently, we must be able to define a general class hierarchy and a general set of operations which can be applied in most applications. How to reach the generality still remain an unsolved issue of the null node method.

7 Bibliography

[1] Herbert Schildt. C++: The Complete Reference. Osborne McGraw-Hill.

[2] *C Program Language FAQ*. URL: http://www.eskimo.com/~scs/C-faq/top.html.

[3] Boddy Schmidt. *The Learning C/C++urve: Getting to the Point(er)*. C/C++ Users Journal, July 1997, page 67.

[4] Boddy Schmidt. *The Learning C/C++urve: Me and My Arrow*. C/C++ Users Journal, August 1997, page 81.

[5] Boddy Schmidt. *The Learning C/C++urve: The Pointer Variations*. C/C++ Users Journal, September 1997, page 79.

[6] Boddy Schmidt. *The Learning C/C++urve: Morte d'Autopointer*. C/C++ Users Journal, February 1998, page 83.

[7] Leendert Ammeraal. Algorithms and Data Structures in C++. Chichester; New York::Wiley 1996

[8] Mark Allen Weiss. Data Structures and Algorithms analysis in C++. Benjamin/Cummings Pub. Co. 1994

[9] Stanley B. Lippman. C++ Primer (2nd Edition). Addison-Wesley Publishing Company 1995.

[10] Scott Robert Ladd. C++ Components and Algorithms (2nd Edition). M&T Books 1994.

Appendix

Code Listing 1: *LBTreeNode.h*

```
// ****************************************************************
// FILE:        LBTreeNode.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Sept. 02. 1997
// DESCRIPTION:  This file contains declarations of three classes:
//              LBaseBTreeNode. LNullBTreeNode and LFullBTreeNode.
//              Both LNullBTreeNode and LFullBTreeNode are
//              derived from LBaseBTreeNode. The three classes
//              are used to describe a binary search tree data structure.
//              The empty node of the bianry tree is not
//              represented by a NULL pointer as in the
//              traditional method. but is represented by a
//              LNullBTreeNode object.
// ****************************************************************

#ifndef LBTREENODE_H
#define LBTREENODE_H

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <conio.h>

#include "LTypes.h"

// ------------------------------------------------
// Class LBaseBTreeNode defines a set of pure
// virtual funcitons to manipulate the binary
// tree node. Here we acutually view each node as
// a valid binary search tree.
// ------------------------------------------------
class LBaseBTreeNode
{
// ---------------------------
// constructors. destructors
// and enum types.
// ---------------------------
public:
  enum
  {
    outputWidth_E= 12 // Width used to format the output of the node data
  };

// ------------------------
// Methods
// ------------------------
public:
  // Insert a new data item into the tree. Return the new
  // root for the binary search tree.
  virtual LBaseBTreeNode* insert(const int data) = 0;

  // Remove a data item from the tree. Return the new root
  // for the binary search tree. If the data item is successfully
  // removed. bRemoveOK is set to TRUE. otherwise bRemoveOK
  // is set to FALSE. If this node iteself need to be removed.
  // bDeleteThis is set to TRUE. otherwise it is set to FALSE.
  virtual LBaseBTreeNode* remove(const int data. BOOL& bDeleteThis. BOOL& bRemoveOK) = 0;

  // Search a data item in the tree. If find. return TRUE.
```

```
  //Otherwise  return FALSE.
  virtual BOOL search(const int data) = 0;

  // Traverse the entire binary search tree using the depth first
  // method. Print the result to the standard output.
  virtual void traverse(void) const = 0;

  // Traverse the entire binary search tree using the depth first
  // method. Print the result to a file represented by the
  // ofstream.
  virtual void traverse(ofstream& ofs) const = 0;

  // Return TRUE if this node is a null node. Otherwise
  // return FALSE.
  virtual BOOL isNull(void) const = 0;

  // Return the node data.
  virtual int getSelfData(void) const = 0;

  // Link the left subtree to a new node.
  virtual void linkLeftSubtree(LBaseBTreeNode* pLeft) = 0;

  // Link the right subtree to a new node.
  virtual void linkRightSubtree(LBaseBTreeNode* pRight) = 0;

  // Get the left subtree pointer. The left subtree cannot
  // be modified.
  virtual const LBaseBTreeNode* getLeftSubtree(void) const = 0;

  // Get the right subtree pointer. The right subtree cannot
  // be modified.
  virtual const LBaseBTreeNode* getRightSubtree(void) const = 0;

  // Get the left subtree pointer. The left subtree can
  // be modified.
  virtual LBaseBTreeNode* getLeftSubtree(void) = 0;

  // Get the right subtree pointer. The right subtree can
  // be modified.
  virtual LBaseBTreeNode* getRightSubtree(void) = 0;

  // Print the node data to the standard output.
  virtual void outputSelfData(void) const = 0;

  // Print the node data to a file.
  virtual void outputSelfData(ofstream& ofs) const = 0;

  // Return the height of the tree.
  virtual getHeight(void) const {return -1;}

  // Return the space required for the entire binary search tree.
  virtual long space(void) const = 0;
};


// ----------------------------------------------------
// Class LNullBTreeNode implements a null binary search tree
// node which has no data and subtrees.
// ----------------------------------------------------
class LNullBTreeNode: public LBaseBTreeNode
{

// -------------
// Methods
// -------------
public:

  // Function insert() cannot be inline function
  // as it need to create a LFullBTreeNode. But class
  // LFullBTreeNode is not defined at this point.
```

```
  virtual LBaseBTreeNode* insert(const int data) ;

// inline methods
  virtual LBaseBTreeNode* remove(const int data, BOOL& bDeleteThis, BOOL& bRemoveOK)
  {
    bDeleteThis= FALSE;
    bRemoveOK= FALSE;
    return this;
  }
  virtual BOOL search(const int data) {return FALSE;}
  virtual void traverse(void) const {}
  virtual void traverse(ofstream& ofs) const {}
  virtual BOOL isNull(void) const {return TRUE;}
  virtual int  getSelfData(void) const { cout << "ERROR: cannot get data from a NULL binary tree node." << endl; return -1;}
  virtual void linkLeftSubtree(LBaseBTreeNode* pLeft) { cout << "ERROR: cannot set the left subtree for NULL binary tree node."
<< endl;}
  virtual void linkRightSubtree (LBaseBTreeNode* pRight)  { cout << "ERROR: cannot set the right subtree for NULL binary tree
node." << endl;}
  virtual const LBaseBTreeNode* getLeftSubtree(void) const {cout << "ERROR: cannot get the left tree of a NULL binary tree node."
<< endl;  getch(); return this;}
  virtual const LBaseBTreeNode* getRightSubtree(void) const { cout << "ERROR: cannot get the right tree of a NULL binary tree
node." << endl; getch(); return this;}
  virtual LBaseBTreeNode* getLeftSubtree(void)  {cout << "ERROR: cannot get the left tree of a NULL binary tree node." << endl;
getch(); return this;}
  virtual LBaseBTreeNode* getRightSubtree(void) { cout << "ERROR: cannot get the right tree of a NULL binary tree node." <<
endl; getch(); return this;}
  virtual void outputSelfData(void) const { cout << setw(outputWidth_E) << "NIL";}
  virtual void outputSelfData(ofstream& ofs) const { ofs << setw(outputWidth_E) << "NIL";}
  virtual long space(void) const {return sizeof(*this) ;}
};


// ----------------------------------------------------
// Class LFullBTreeNode implements a full binary tree
// node which has data and subtrees.
// Since a full binary tree node has pointers to subtree
// nodes, a tree node can be used to represent an entire
// binary tree.
// ----------------------------------------------------
class LFullBTreeNode: public LBaseBTreeNode
{

// ------------------------------------------
// Constructors, destructors and enum types.
// ------------------------------------------
public:
  LFullBTreeNode(const int newData);
  LFullBTreeNode(const LFullBTreeNode& rNode);
  virtual ~LFullBTreeNode(void);

// ---------
// Methods
// ---------
public:
  virtual LBaseBTreeNode* insert(const int data) ;
  virtual LBaseBTreeNode* remove(const int data, BOOL& bDeleteThis, BOOL& bRemoveOK);
  virtual BOOL search(const int data);
  virtual void traverse(void) const;
  virtual void traverse(ofstream& ofs) const;

  virtual void removeSubtrees(void);
  virtual long space(void) const;

// operators
  LFullBTreeNode& operator= (const LFullBTreeNode& rNode);

// inline methods
  virtual BOOL    isNull          (void) const { return FALSE;}
  virtual int     getSelfData     (void) const {return m_data;}
  virtual void    linkLeftSubtree (LBaseBTreeNode* pLeft) {m_pLeft= pLeft;}
```

```
  virtual void    linkRightSubtree    (LBaseBTreeNode* pRight) {m_pRight= pRight;}
  virtual void    outputSelfData(void) const { cout << setw(outputWidth_E) << m_data;}
  virtual void    outputSelfData(ofstream& ofs) const { ofs << setw(outputWidth_E) << m_data;}

  virtual const LBaseBTreeNode*   getLeftSubtree(void) const { return m_pLeft;}
  virtual const LBaseBTreeNode*   getRightSubtree(void) const { return m_pRight;}
  virtual LBaseBTreeNode*         getLeftSubtree(void) { return m_pLeft;}
  virtual LBaseBTreeNode*         getRightSubtree(void) { return m_pRight;}

  virtual  LNullBTreeNode*        getStaticNullNode(void)  {return &m_nullNode_S;}
  static   LNullBTreeNode*        getNullNode_S (void) {return &m_nullNode_S;}

protected:
  virtual void setSelfData(const int data) {m_data= data;}

private:
  LFullBTreeNode* detachSmallestNode(void);

// ------------
// Attributes
// ------------
private:
  int                 m_data;       // data kept with the node
  LBaseBTreeNode  *m_pLeft,     // left child. may point to a LNullBTreeNode or LFullBTreeNode
                  *m_pRight;    // right child.

  static LNullBTreeNode m_nullNode_S; // used to represent the leaf. It is
                                      // is static since all the leaves can
                                      // use the same null node.
protected:
  static BOOL   m_bCountNullNodeSpace_S; // TRUE if the space occupied by the null node is counted.
};

#endif //LBTREENODE_H
```

Code Listing 2: *LBTreeNode.cpp*

```
// ***************************************************************
// FILE:        LBTreeNode.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Sept. 02. 1997
// DESCRIPTION:  Refer to LBTreeNode.h
// ***************************************************************

// -------------
// Includes
// -------------

#include <iostream.h>
#include <assert.h>

#include "LTypes.h"
#include "LBTreeNode.h"


// ------------------------------------------------
// Initialization of the static member variables.
// ------------------------------------------------

LNullBTreeNode LFullBTreeNode::m_nullNode_S;
BOOL LFullBTreeNode::m_bCountNullNodeSpace_S= FALSE;

// ------------------------------------------------------------
// Function:       LFullBTreeNode()
// Descritpion:    Constructor.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
LFullBTreeNode::LFullBTreeNode
  (const int newData)
{
  m_data= newData;
  m_pLeft= getStaticNullNode();
  m_pRight= getStaticNullNode();
}


// ------------------------------------------------------------
// Function:       LFullBTreeNode()
// Descritpion:    Copy constructor.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
LFullBTreeNode::LFullBTreeNode
  (const LFullBTreeNode& rNode)
{
  *this= rNode;
}


// ------------------------------------------------------------
// Function:       ~LFullBTreeNode()
// Descritpion:    Destructor.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
LFullBTreeNode::~LFullBTreeNode
  (void)
{
}


// ------------------------------------------------------------
```

```
// Function:        removeSubtrees()
// Description:     Remove all the subtrees of the tree node and
//                  set the left and the right subtree node to be
//                  null nodes. Memory allocated for the subtrees
//                  are freed.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------------
void LFullBTreeNode::removeSubtrees
  (void)
{
  // ------------------------
  // Delete the left subtree
  // ------------------------

  if (!m_pLeft->isNull())
  {
    if (m_pLeft->getLeftSubtree()->isNull() && m_pLeft->getRightSubtree()->isNull())
    {
      // m_pLeft is a leaf now. we can delete it.
      delete m_pLeft;
      m_pLeft= getStaticNullNode();
    }
    else
    {
      // Recursively delete the left subtree.
      ((LFullBTreeNode*) m_pLeft)->removeSubtrees();
    }
  }

  // -------------------------
  // Delete the right subtree
  // -------------------------

  if (!m_pRight->isNull())
  {
    if (m_pRight->getLeftSubtree()->isNull() && m_pRight->getRightSubtree()->isNull())
    {
      // m_pRight is a leaf now. we can delete it.
      delete m_pRight;
      m_pRight= getStaticNullNode();
    }
    else
    {
      // Recursively delete the right subtree.
      ((LFullBTreeNode*) m_pRight)->removeSubtrees();
    }
  }

  // ----------------------------------------------------------
  // By recursively calling removeSubtrees() for the left node and
  // the right node. the two nodes will be turned into leaves.
  // We can delete them now.
  // ----------------------------------------------------------

  if (!m_pLeft->isNull())
  {
    assert(m_pLeft->getLeftSubtree()->isNull());
    assert(m_pLeft->getRightSubtree()->isNull());
    delete m_pLeft;
    m_pLeft= getStaticNullNode();
  }

  if (!m_pRight->isNull())
  {
    assert(m_pRight->getLeftSubtree()->isNull());
    assert(m_pRight->getRightSubtree()->isNull());
    delete m_pRight;
    m_pRight= getStaticNullNode();
  }
```

```
  assert(m_pLeft->isNull() && m_pRight->isNull());
  return;

}

// ------------------------------------------------------------
// Function:       insert()
// Description:    Insert a data item into the binary tree.
//                 Return the new root for the binary tree.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
LBaseBTreeNode* LFullBTreeNode::insert
  (const int data) // new data item to be inserted
{
  if (data < m_data)
  {
    m_pLeft= m_pLeft->insert(data); // recursive function call
  }
  else if (data > m_data)
  {
    m_pRight= m_pRight->insert(data); // recursive function call
  }
  // else {m_data = data} we don't allow duplication of data

  return this;
}

// ------------------------------------------------------------------
// Function:       remove()
// Description:    Remove a data item from the binary tree.
//                 Return the new root for the binary tree.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------------
LBaseBTreeNode* LFullBTreeNode::remove
  (const int    data, // data to be removed
   BOOL&  bDeleteThis, // TRUE if this node itself need to be deleted.
   BOOL&  bRemoveOK)  // TRUE if we sucessfully remove the data item from the tree
{

  LBaseBTreeNode *pNewRoot;

  bDeleteThis= FALSE;
  bRemoveOK= FALSE;

  if ( data == m_data)
  {
    // ---------------------------------------------
    // This node is the node which contains the
    // data item to be removed. Set bDeleteThis to
    // TRUE and bRemoveOK to TRUE.
    // ---------------------------------------------
    bDeleteThis= TRUE;
    bRemoveOK= TRUE;

    if (m_pLeft->isNull())
    {
      pNewRoot= m_pRight; // new root is the right subtree.
    }
    else if (m_pRight->isNull())
    {
      pNewRoot= m_pLeft; // new root is the left subtree.
    }
    else
    {
      // ---------------------------------
      // None of the left and right subtree
      // is Null, we should replace the root
```

```
      // with the smallest node of the right
      // subtree.
      // ------------------------------------

      LFullBTreeNode* pNewThisNode;

      pNewThisNode= ((LFullBTreeNode*) m_pRight)->detachSmallestNode();

      // Connect the new root with the left subtree.
      pNewThisNode->linkLeftSubtree(m_pLeft);

      pNewRoot= pNewThisNode;
    }
  }
  else if ( data < m_data)
  {
    BOOL bDeleteLeft;
    LBaseBTreeNode* pOldLeft= m_pLeft;

    m_pLeft= m_pLeft->remove(data, bDeleteLeft, bRemoveOK);
    if (bDeleteLeft)
    {
      delete pOldLeft;
    }

    // Data item is removed from the left subtree. The tree
    // root is still this node.
    pNewRoot= this;
  }
  else if (data > m_data)
  {
    BOOL bDeleteRight;
    LBaseBTreeNode* pOldRight= m_pRight;

    m_pRight= m_pRight->remove(data, bDeleteRight, bRemoveOK);
    if (bDeleteRight)
    {
      delete pOldRight;
    }

    // Data item is removed from the right subtree. The tree
    // root is still this node.
    pNewRoot= this;
  }

  return pNewRoot;
}


// ------------------------------------------------------------
// Function:       detachSmallestNode()
// Descritpion:    Detach the smallest (deepest left child) node
//                 from the binary tree and re-connect the
//                 rest of the binary tree. The detached node is
//                 returned.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
LFullBTreeNode* LFullBTreeNode::detachSmallestNode
  (void)
{
  if (m_pLeft->isNull())
  {
    // ------------------------------------------------
    // Since all the data items in the right subtree
    // is greater than m_data, the smallest node is
    // this node.
    // ------------------------------------------------
    return this;
  }
```

```
    // ---------------------------------------------------
    // Find the leftmost node which is the smallest node.
    // ---------------------------------------------------

    LFullBTreeNode* pSmallest= (LFullBTreeNode*) m_pLeft;
    LFullBTreeNode* pSmallestParent= this;

    while (!pSmallest->getLeftSubtree()->isNull())
    {
      pSmallestParent= pSmallest;
      pSmallest= (LFullBTreeNode*) pSmallest->getLeftSubtree();
    }

    // ---------------------------------------------------
    // Detach the smallest node from the tree. Reconnect
    // the rest of the tree. Return the detached node.
    // ---------------------------------------------------

    pSmallestParent->linkLeftSubtree(pSmallest->getRightSubtree());
    pSmallest->linkRightSubtree(this);
    return pSmallest;
}

// ------------------------------------------------------------
// Function:        search()
// Description:     Return TRUE if find data in the tree. Otherwise
//                  return FALSE.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
BOOL LFullBTreeNode::search
  (const int data)
{
  BOOL bRet= FALSE;

  if (data == m_data)
  {
    bRet= TRUE;
  }
  else if (data < m_data)
  {
    bRet= m_pLeft->search(data);
  }
  else if (data > m_data)
  {
    bRet= m_pRight->search(data);
  }

  return bRet;
}


// ------------------------------------------------------------
// Function:        traverse()
// Description:     Output the data in the binary tree to standard
//                  output recursively. Use depth first method.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
void LFullBTreeNode::traverse
  (void) const
{

  m_pLeft->traverse();

  m_pLeft->outputSelfData();
  cout << setw(outputWidth_E) << "<======";
  cout << setw(outputWidth_E) << m_data;
  cout << setw(outputWidth_E) << "======>";
```

```
  m_pRight->outputSelfData();
  cout << endl;

  m_pRight->traverse();
}

// ---------------------------------------------------------------
// Function:       traverse()
// Description:    Output the data in the binary tree to a file
//                 output stream recursively. Use depth first method.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
void LFullBTreeNode::traverse
  (ofstream& ofs) const
{
  m_pLeft->traverse(ofs);

  m_pLeft->outputSelfData(ofs);
  ofs << setw(outputWidth_E) << "<======";
  ofs << setw(outputWidth_E) << m_data;
  ofs << setw(outputWidth_E) << "======>";
  m_pRight->outputSelfData(ofs);
  ofs << endl;

  m_pRight->traverse(ofs);
}

// ---------------------------------------------------------------
// Function:       operator=()
// Description:    Copy the entire binary tree, including the
//                 subtrees. New memory is allocated.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
LFullBTreeNode& LFullBTreeNode:: operator=
  (const LFullBTreeNode& rNode)
{

  static int flag= 0;
  // ---------------------------
  // Delete the existing tree.
  // ---------------------------

  if (flag == 0)
  {
    removeSubtrees();
    flag= 1;
    m_data= rNode.getSelfData();
  }

  if (!rNode.getLeftSubtree()->isNull())
  {
    m_pLeft= new LFullBTreeNode(rNode.getLeftSubtree()->getSelfData());
    (*((LFullBTreeNode*) m_pLeft))= (*((LFullBTreeNode *)rNode.getLeftSubtree()));
  }

  if (!rNode.getRightSubtree()->isNull())
  {
    m_pRight= new LFullBTreeNode(rNode.getRightSubtree()->getSelfData());
    (*((LFullBTreeNode*) m_pRight))= (*((LFullBTreeNode*)rNode.getRightSubtree()));
  }

  flag= 0;

  return *this;
}

// ---------------------------------------------------------------
// Function:       space()
```

```
// Description:  Return the space of the entire binary tree.
//               Note that in an application, all the empty
//               nodes in all the binary trees are actually
//               represented by the same static null node.
//               Therefore we only calculate the space
//               occupied by the null node once in the
//               application.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
long LFullBTreeNode::space
  (void) const
{
  long treeSpace= sizeof(*this);
  if (!m_pLeft->isNull())
  {
    treeSpace+= m_pLeft->space();
  }

  if (!m_pRight->isNull())
  {
    treeSpace+= m_pRight->space();
  }

  if (!m_bCountNullNodeSpace_S)
  {
    treeSpace+= m_nullNode_S.space();
    m_bCountNullNodeSpace_S= TRUE;
  }
  return treeSpace;
}


////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
// Implementation for class LNullBTreeNode
//

// ---------------------------------------------------------------
// Function:      insert
// Description:   Insert a data item into the null node.
//                Return a full node.
// Errors Generated: None
// Limitations:   Didn't handle the memory allocation failure
//                if occurred.
// ---------------------------------------------------------------
LBaseBTreeNode* LNullBTreeNode::insert
  (const int data)
{
  return new LFullBTreeNode(data);
}
```

Code Listing 3: *LBTree.h*

```
// ****************************************************************
// FILE:            LBTree.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Sept. 15, 1997
// DESCRIPTION:  LBTree encapsulates a binary tree data structure.
//                      It uses a LBaseBTreeNode object as its root.
// ****************************************************************

#ifndef LBTREE_H
#define LBTREE_H

#include <fstream.h>

#include "LTypes.h"
#include "LBTreeNode.h"
#include "LAVLBTreeNode.h"

class LBTree
{
// ------------------------------------
// Constructors, destructors and
// enum types
// ------------------------------------

public:
  LBTree (void);
  LBTree (const LBTree& rTree) {*this= rTree;}
  virtual ~LBTree(void);

// -------------------------
// Member functions
// -------------------------
public:
  BOOL  remove (const int data);
  void  empty(void);

// operators
  LBTree& operator= (const LBTree& rTree);

// inline functions
  void  insert    (const int data) {m_pRoot= m_pRoot->insert(data);}
  void  traverse  (void) const {m_pRoot->traverse();}
  void  traverse  (ofstream& ofs) const {m_pRoot->traverse(ofs);}
  BOOL  search    (const int data) const {return m_pRoot->search(data);}
  long  space     (void)  { return (sizeof(*this) + m_pRoot->space());}

  const LBaseBTreeNode& getRoot(void) const {return *m_pRoot;}

protected:
  virtual void initRoot(void);
  void setRoot(LBaseBTreeNode* pRoot) { m_pRoot= pRoot;}

// ----------------------
// Attributes
// ----------------------
protected:
  LBaseBTreeNode* m_pRoot;
};


class LAVLBTree : public LBTree
{
public:
  LAVLBTree(void);
  LAVLBTree(const LAVLBTree& rTree) {*this= rTree;}
```

```
public:
  LAVLBTree& operator= (const LAVLBTree& rTree);


protected:
  virtual void initRoot(void);
};

#endif
```

Code Listing 4: *LBTree.cpp*

```
// ****************************************************************
// FILE:            LBTree.cpp
// AUTHOR:          Zhilin Li
// CREATION DATE: Sept. 15, 1997
// DESCRIPTION:   Refer to LBTree.h
// ****************************************************************

#include <assert.h>
#include "LBTree.h"
#include "LBTreeNode.h"

// ----------------------------------------------------------
// Function:        LBTree()
// Descritpion:   Default constructor. Initialize the root
//                  to be the static LNullBTreeNode member
//                  variable defined in LFullBTreeNode class.
//                  class.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LBTree::LBTree
  (void)
{
  initRoot();
}


// ----------------------------------------------------------
// Function:        ~LBTree()
// Descritpion:   Destructor. Delete the entire binary tree.
//                  Free memory allocated.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LBTree::~LBTree(void)
{
  empty();
}

// ----------------------------------------------------------
// Function:        remove()
// Descritpion:   Remove data from the binary tree. Return
//                  TRUE if found the data and successfully
//                  remove it. Otherwise return FALSE.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
BOOL LBTree::remove
  (const int data)
{
  BOOL bDeleteRoot,
     bRemoveOK;

  LBaseBTreeNode* pOldRoot= m_pRoot;

  m_pRoot= m_pRoot->remove(data, bDeleteRoot, bRemoveOK);

  if (bDeleteRoot)
  {
    delete pOldRoot;
  }

  return bRemoveOK;
}
```

```
// ---------------------------------------------------------
// Function:      operator= ()
// Description:   Copy the entire binary tree. Allocate
//                memory as needed.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
LBTree& LBTree::operator=
  (const LBTree& rTree)
{
  if (m_pRoot->isNull())
  {
    if (!rTree.getRoot().isNull())
    {
      m_pRoot= new LFullBTreeNode(rTree.getRoot().getSelfData());
      (*((LFullBTreeNode*) m_pRoot))= ((LFullBTreeNode&)rTree.getRoot());
    }
  }
  else
  {
    if (rTree.getRoot().isNull())
    {
      ((LFullBTreeNode*) m_pRoot)->removeSubtrees();
      delete m_pRoot;

      m_pRoot= LFullBTreeNode::getNullNode_S();
    }
    else
    {
      (*((LFullBTreeNode*) m_pRoot))= ((LFullBTreeNode& )rTree.getRoot());
    }
  }

  return *this;
}

// ---------------------------------------------------------
// Function:      emptyTree()
// Description:   Empty the entire tree. Set the root to
//                be a null tree node.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
void LBTree::empty
  (void)
{
  if (!m_pRoot->isNull())
  {
    ((LFullBTreeNode*) m_pRoot)->removeSubtrees();
    delete m_pRoot;

    initRoot();
  }
}


// ---------------------------------------------------------
// Function:      initRoot()
// Description:   Set the root to the null binary tree node.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
void LBTree::initRoot
  (void)
{
  setRoot(LFullBTreeNode::getNullNode_S());
}

//////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////////
```

```
// Implementation of LAVLBTree
//

// ----------------------------------------------------------
// Function:       initRoot()
// Description:    Set the root to the null AVL tree node
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
void LAVLBTree::initRoot
  (void)
{
  setRoot(LFullAVLBTreeNode::getNullNode_S());
}



// ----------------------------------------------------------
// Function:       LAVLBTree
// Description:    Constructor. Intialize the root.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------

LAVLBTree::LAVLBTree
  (void)
{
  initRoot();
}

// ----------------------------------------------------------
// Function:       operator= ()
// Description:  Copy the entire AVL binary tree. Allocate
//                 memory as needed.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LAVLBTree& LAVLBTree::operator=
  (const LAVLBTree& rTree)
{
  if (m_pRoot->isNull())
  {
    if (!rTree.getRoot().isNull())
    {
      m_pRoot= new LFullAVLBTreeNode(rTree.getRoot().getSelfData());
      (*((LFullAVLBTreeNode*) m_pRoot))= ((LFullAVLBTreeNode&) rTree.getRoot());
    }
  }
  else
  {
    if (rTree.getRoot().isNull())
    {
      ((LFullBTreeNode*) m_pRoot)->removeSubtrees();
      delete m_pRoot;

      m_pRoot= LFullAVLBTreeNode::getNullNode_S();
    }
    else
    {
      (*((LFullAVLBTreeNode*) m_pRoot))= ((LFullAVLBTreeNode&) rTree.getRoot());
    }
  }

  return *this;
}
```

Code Listing 5: *LPtrBTreeNode.h*

```
// ***************************************************************
// FILE:            LPtrBTreeNode.h
// AUTHOR:          Zhilin Li
// CREATION DATE: Oct. 07, 1997
// DESCRIPTION:   Class LPtrBTreeNode implements a binary tree
//                  node. The node itself actually represents a
//                  valid binary tree. The empty nodes of the tree
//                  are represented using NULL pointers.
// ***************************************************************

#ifndef LPTRBTREENODE_H
#define LPTRBTREENODE_H

#include <fstream.h>
#include "LTypes.h"

class LPtrBTreeNode
{
// ------------------------------------------
// Constructors, destructors and enum types.
// ------------------------------------------
public:
  LPtrBTreeNode(const int data);
  LPtrBTreeNode(const LPtrBTreeNode& rNode);
  virtual ~LPtrBTreeNode(void);

  enum
  {
    outputWidth_E= 12
  };

// ------------------
// Methods
// ------------------
public:
  virtual LPtrBTreeNode* insert(const int data);
  virtual LPtrBTreeNode* remove(const int data, BOOL& bDeleteThis, BOOL& bRemoveOK);
  virtual void traverse(void) const;
  virtual void traverse(ofstream& ofs) const;
  virtual void removeSubtrees(void);

  virtual long space(void) const;
  BOOL search(const int data);

// operators
  LPtrBTreeNode& operator = (const LPtrBTreeNode& rNode);

// inline methods
  void                 linkLeftSubtree  (LPtrBTreeNode* pLeft) {m_pLeft= pLeft;}
  void                 linkRightSubtree (LPtrBTreeNode* pRight) {m_pRight= pRight;}
  LPtrBTreeNode*       getLeftSubtree   (void) {return m_pLeft;}
  LPtrBTreeNode*       getRightSubtree  (void) {return m_pRight;}
  const LPtrBTreeNode* getLeftSubtree   (void) const {return m_pLeft;}
  const LPtrBTreeNode* getRightSubtree  (void) const {return m_pRight;}
  int                  getSelfData      (void) const {return m_data;}
  void                 setSelfData      (const int data) {m_data= data;}

private:
  LPtrBTreeNode* detachSmallestNode(void);

// ----------------
// Attributes
// ----------------
private:
  int m_data;
  LPtrBTreeNode* m_pLeft;
```

```
  LPtrBTreeNode* m_pRight;
};
#endif
```

Code Listing 6: *LPtrBTreeNode.cpp*

```
// ****************************************************************
// FILE:        LPtrBTreeNode.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Oct. 07, 1997
// DESCRIPTION:  Refer to LPtrBTreeNode.h
// ****************************************************************

#include <assert.h>
#include <iostream.h>
#include <iomanip.h>

#include "LPtrBTreeNode.h"

// ----------------------------------------------------------
// Function:       LPtrBTreeNode()
// Descritpion:    Constructor.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LPtrBTreeNode::LPtrBTreeNode
  (const int newData)
{
  m_data= newData;
  m_pLeft= NULL;
  m_pRight= NULL;
}

// ----------------------------------------------------------
// Function:       LPtrBTreeNode()
// Descritpion:    Copy constructor.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LPtrBTreeNode::LPtrBTreeNode
  (const LPtrBTreeNode& rNode)
{
  *this= rNode;
}

// ----------------------------------------------------------
// Function:       ~LPtrBTreeNode()
// Descritpion:    Destructor.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LPtrBTreeNode::~LPtrBTreeNode
  (void)
{
}


// ----------------------------------------------------------
// Function:       insert()
// Descritpion:    Insert a data item into the binary tree.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LPtrBTreeNode* LPtrBTreeNode::insert
  (const int data)
{
  if (data < m_data)
  {
    if (m_pLeft != NULL)
    {
      m_pLeft->insert(data);
    }
```

```
    else
    {
      m_pLeft= new LPtrBTreeNode(data);
    }
  }
  else if (data > m_data)
  {
    if (m_pRight != NULL)
    {
      m_pRight->insert(data);
    }
    else
    {
      m_pRight= new LPtrBTreeNode(data);
    }
  }

  return this;
}


// ------------------------------------------------------------
// Function:       remove()
// Descritpion:    Remove a data item from the binary tree.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
LPtrBTreeNode* LPtrBTreeNode::remove
  (const int data,    // data to be removed
   BOOL& bDeleteThis, // TRUE if this node itself need to be deleted
   BOOL& bRemoveOK)   // TRUE if the data item is successfully removed from the binary tree.
{
  bDeleteThis= FALSE;
  bRemoveOK= FALSE;

  if (data == m_data)
  {
    bDeleteThis= TRUE;
    bRemoveOK= TRUE;
    if (m_pLeft == NULL)
    {
      return m_pRight;
    }
    else if (m_pRight == NULL)
    {
      return m_pLeft;
    }
    else
    {
      LPtrBTreeNode* pNewThisNode= m_pRight->detachSmallestNode();

      pNewThisNode->linkLeftSubtree(m_pLeft);

      if (pNewThisNode != m_pRight)
      {
        pNewThisNode->linkRightSubtree(m_pRight);
      }

      return pNewThisNode;
    }
  }
  else if ( data < m_data)
  {
    if (m_pLeft == NULL)
    {
      // We cannot find the data item. bRemoveOK
      // and bDeleteThis are set to FALSE already
      // Root is still this node, so return this.
      return this;
    }
```

```
    BOOL bDeleteLeft;
    LPtrBTreeNode* pOldLeft= m_pLeft;

    m_pLeft= m_pLeft->remove(data, bDeleteLeft, bRemoveOK);
    if (bDeleteLeft)
    {
      delete pOldLeft;
    }
    return this;
  }
  else if (data > m_data)
  {
    if (m_pRight == NULL)
    {
      // We cannot find the data item, bRemoveOK
      // and bDeleteThis are set to FALSE alreay.
      // Root is still this node, so return this.
      return this;
    }
    BOOL bDeleteRight;
    LPtrBTreeNode* pOldRight= m_pRight;

    m_pRight= m_pRight->remove(data, bDeleteRight, bRemoveOK);
    if (bDeleteRight)
    {
      delete pOldRight;
    }
    return this;
  }

  return this;
}

// ------------------------------------------------------------------
// Function:       search()
// Descritpion:    Search data in the binary tree. If found,
//                 return TRUE, otherwise return FALSE.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------------
BOOL LPtrBTreeNode::search
  (const int data)
{
  BOOL bRet= FALSE;

  if (data == m_data)
  {
    bRet= TRUE;
  }
  else if (data < m_data)
  {
    if (m_pLeft != NULL)
      bRet= m_pLeft->search(data);
  }
  else if (data > m_data)
  {
    if (m_pRight != NULL)
      bRet= m_pRight->search(data);
  }

  return bRet;
}

// ------------------------------------------------------------------
// Function:       traverse()
// Descritpion:    Output the data in the binary tree to the
//                 standard output recursively.
// Errors Generated: None
// Limitations:
```

```
// ---------------------------------------------------------------
void LPtrBTreeNode::traverse
  (void) const
{
  if (m_pLeft != NULL)
  {
    m_pLeft->traverse();
    cout << setw(outputWidth_E) << m_pLeft->getSelfData();
  }
  else
  {
    cout << setw(outputWidth_E) << "NIL";
  }

  cout << setw(outputWidth_E) << "<======";
  cout << setw(outputWidth_E) << m_data;
  cout << setw(outputWidth_E) << "======>";

  if (m_pRight != NULL)
  {
    cout << setw(outputWidth_E) << m_pRight->getSelfData() << endl;
    m_pRight->traverse();
  }
  else
  {
    cout << setw(outputWidth_E) << "NIL" << endl;
  }

}


// ---------------------------------------------------------------
// Function:       traverse()
// Descritpion:    Output the data in the binary tree to a
//                 file output stream recursively.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
void LPtrBTreeNode::traverse
  (ofstream& ofs) const
{
  if (m_pLeft != NULL)
  {
    m_pLeft->traverse(ofs);
    ofs << setw(outputWidth_E) << m_pLeft->getSelfData();
  }
  else
  {
    ofs << setw(outputWidth_E) << "NIL";
  }

  ofs << setw(outputWidth_E) << "<======";
  ofs << setw(outputWidth_E) << m_data;
  ofs << setw(outputWidth_E) << "======>";

  if (m_pRight != NULL)
  {
    ofs << setw(outputWidth_E) << m_pRight->getSelfData() << endl;
    m_pRight->traverse(ofs);
  }
  else
  {
    ofs << setw(outputWidth_E) << "NIL" << endl;
  }

}

// ---------------------------------------------------------------
// Function:       operator=()
// Descritpion:    Copy the entire binary tree, including the
```

```
//              subtrees. New memory is allocated to copy
//              the entire tree structure.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
LPtrBTreeNode& LPtrBTreeNode::operator =
  (const LPtrBTreeNode& rNode)
{

  static int flag= 0;

  // ----------------------------
  // Delete the existing tree.
  // ----------------------------

  if (flag == 0)
  {
    removeSubtrees();
    flag ++;
    m_data= rNode.getSelfData();
  }

  if (rNode.getLeftSubtree() != NULL)
  {
    m_pLeft= new LPtrBTreeNode(rNode.getLeftSubtree()->getSelfData());
    *m_pLeft= *rNode.getLeftSubtree();
  }

  if (rNode.getRightSubtree() != NULL)
  {
    m_pRight= new LPtrBTreeNode(rNode.getRightSubtree()->getSelfData());
    *m_pRight= *rNode.getRightSubtree();
  }

  flag= 0;

  return *this;

}


// ---------------------------------------------------------------
// Function:       removeSubtrees()
// Descritpion:    Remove all the subtrees of the tree node and
//              set the left and the right subtrees to be NULL.
//              Free the memory allocated for the subtrees.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
void LPtrBTreeNode::removeSubtrees
  (void)
{
  // ------------------------
  // Delete the left subtree
  // ------------------------

  if (m_pLeft != NULL)
  {
    if ((m_pLeft->getLeftSubtree() == NULL) && (m_pLeft->getRightSubtree() == NULL))
    {
      // m_pLeft is a leaf. we can delete it.
      delete m_pLeft;
      m_pLeft= NULL;
    }
    else
    {
      m_pLeft->removeSubtrees();
    }
  }
```

```
  // ---------------------------
  // Delete the right subtree
  // ---------------------------

  if (m_pRight != NULL)
  {
    if ((m_pRight->getLeftSubtree() == NULL) && (m_pRight->getRightSubtree() == NULL))
    {
      // m_pRight is a leaf. we can delete it.
      delete m_pRight;
      m_pRight= NULL;
    }
    else
    {
      m_pRight->removeSubtrees();
    }
  }

  // ---------------------------------------------------------------
  // By recursively calling removeSubtrees() for the left node and
  // the right node. the two nodes become leaves. we can delete
  // them.
  // ---------------------------------------------------------------

  if (m_pLeft != NULL)
  {
    assert(m_pLeft->getLeftSubtree() == NULL);
    assert(m_pLeft->getRightSubtree() == NULL);
    delete m_pLeft;
    m_pLeft= NULL;
  }

  if (m_pRight != NULL)
  {
    assert(m_pRight->getLeftSubtree() == NULL);
    assert(m_pRight->getRightSubtree() == NULL);
    delete m_pRight;
    m_pRight= NULL;
  }

  assert((m_pLeft == NULL) && (m_pRight == NULL));

  return;

}

// ---------------------------------------------------------------
// Function:       detachSmallestNode()
// Description:    Detach the smallest (deepest left child) node
//                 from the binary tree and re-connect the
//                 rest of the binary tree. Return the detached
//                 smallest node.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
LPtrBTreeNode* LPtrBTreeNode::detachSmallestNode
  (void)
{
  if (m_pLeft == NULL)
  {
    return this;
  }
  else
  {
    LPtrBTreeNode* pSmallest= m_pLeft;
    LPtrBTreeNode* pSmallestParent= this;

    while (pSmallest->getLeftSubtree() != NULL)
    {
      pSmallestParent= pSmallest;
```

```
      pSmallest= pSmallest->getLeftSubtree();
    }

    pSmallestParent->linkLeftSubtree(pSmallest->getRightSubtree());
    return pSmallest;
  }
}


// ------------------------------------------------------------------
// Function:        space()
// Description:     Return the space needed for the entire tree.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------------
long LPtrBTreeNode::space
  (void) const
{

  long treeSpace= sizeof(*this);

  if (m_pLeft != NULL)
  {
    treeSpace+= m_pLeft->space();
  }

  if (m_pRight != NULL)
  {
    treeSpace+= m_pRight->space();
  }

  return treeSpace;
}
```

Code Listing 7: *LPtrBTree.h*

```
// ************************************************************
// FILE:        LPtrBTree.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Oct. 15, 1997
// DESCRIPTION:  LPtrBTree encapsulates a binary tree data structure.
//              It uses a LPtrBTreeNode pointer as its root.
//              LPtrAVLBTree encapsulates a AVL binary tree data
//              structure. It uses a LPtrAVLBTreeNode pointer
//              as its root.
// ************************************************************

#ifndef LPTRBTREE_H
#define LPTRBTREE_H

#include "LTypes.h"
#include "LPtrBTreeNode.h"

class LPtrBTree
{
// ----------------------------
// Constructors, destructors and
// enum types
// ----------------------------

public:

  LPtrBTree(void);
  LPtrBTree(const LPtrBTree& rTree) {*this = rTree;}
  virtual ~LPtrBTree(void);

// -------------------------
// Member functions
// -------------------------
public:
  virtual void insert (const int data);
  BOOL remove (const int data);
  void empty(void);
  BOOL search (const int data) const;

// operators
  LPtrBTree& operator = (const LPtrBTree& rTree);

// inline functions
  void traverse   (void) const { if (m_pRoot != NULL) m_pRoot->traverse();}
  void traverse   (ofstream& ofs) const { if (m_pRoot != NULL) m_pRoot->traverse(ofs);}
  long space      (void) const { if (m_pRoot != NULL) return (sizeof(*this) + m_pRoot->space()); else return sizeof(*this);}

  const LPtrBTreeNode* getRoot(void) const {return m_pRoot;}
  LPtrBTreeNode*       getRoot(void)       {return m_pRoot;}
  void                 setRoot(LPtrBTreeNode* pRoot) {m_pRoot= pRoot;}

// ----------------------
// Attributes
// ----------------------
private:
  LPtrBTreeNode* m_pRoot;

};

class LPtrAVLBTree : public LPtrBTree
{
public:
  LPtrAVLBTree(void) {}
  LPtrAVLBTree(const LPtrAVLBTree& rTree) {*this = rTree;}

public:
```

```
  virtual void insert (const int data);

  // operators
  LPtrAVLBTree& operator = (const LPtrAVLBTree& rTree);
};

#endif // LPTRBTREE_H
```

Code Listing 8: *LPtrBTree.cpp*

```
// ****************************************************************
// FILE:        LPtrBTree.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Oct. 15, 1997
// DESCRIPTION:  Refer to LPtrBTree.h
// ****************************************************************

#include <stddef.h>
#include <assert.h>
#include "LPtrBTree.h"
#include "LPtrBTreeNode.h"
#include "LPtrAVLBTreeNode.h"

// ----------------------------------------------------------
// Function:       LPtrBTree()
// Descritpion:    Default constructor.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LPtrBTree::LPtrBTree
  (void)
{
  m_pRoot= NULL;
}



// ----------------------------------------------------------
// Function:       ~LPtrBTree()
// Descritpion:    Destructor. Delete the root node.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LPtrBTree::~LPtrBTree(void)
{
  empty();
}

// ----------------------------------------------------------
// Function:       insert()
// Descritpion:    insert new data into the binary tree.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
void LPtrBTree::insert
  (const int data)
{
  if (m_pRoot == NULL)
  {
    m_pRoot= new LPtrBTreeNode(data);
  }
  else
  {
    m_pRoot= m_pRoot->insert(data);
  }
}

// ----------------------------------------------------------
// Function:       remove()
// Descritpion:    Remove data from the binary tree. Return
//                 TRUE if found the data and successfully
//                 remove it. Otherwise return FALSE.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
BOOL LPtrBTree::remove
```

```
  (const int data)
{
  if (m_pRoot == NULL)
    return FALSE;

  BOOL bDeleteRoot,
     bRemoveOK;

  LPtrBTreeNode* pOldRoot= m_pRoot;

  m_pRoot= m_pRoot->remove(data, bDeleteRoot, bRemoveOK);

  if (bDeleteRoot)
  {
    delete pOldRoot;
  }

  return bRemoveOK;
}

// ---------------------------------------------------------
// Function:       operator=()
// Descritpion:    Copy the entire binary tree.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
LPtrBTree& LPtrBTree::operator =
  (const LPtrBTree& rTree)
{
  if (m_pRoot == NULL)
  {
    if (rTree.getRoot() != NULL )
    {
      m_pRoot= new LPtrBTreeNode(rTree.getRoot()->getSelfData());
      *m_pRoot= *rTree.getRoot();
    }
  }
  else
  {
    if (rTree.getRoot() == NULL)
    {
      m_pRoot->removeSubtrees();
      delete m_pRoot;
      m_pRoot= NULL;
    }
    else
    {
      *m_pRoot= *rTree.getRoot();
    }
  }

  return *this;
}

// ---------------------------------------------------------
// Function:       empty()
// Descritpion:    Empty the entire tree. Set the root to
//                 be a NULL pointer
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
void LPtrBTree::empty
  (void)
{
  if (m_pRoot != NULL)
  {
    m_pRoot->removeSubtrees();
    delete m_pRoot;
    m_pRoot= NULL;
  }
```

```
}

// ---------------------------------------------------------
// Function:       search()
// Description:    Check if data is contained in the binary
//                 tree. If yes, return TRUE. Otherwise return
//                 FALSE.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
BOOL LPtrBTree::search
  (const int data) const
{
  if (m_pRoot == NULL)
    return FALSE;
  else
    return m_pRoot->search(data);
}


//////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
// Implementation of LPtrAVLBTree
//

// ---------------------------------------------------------
// Function:       insert()
// Description:    insert new data into the AVL binary tree.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
void LPtrAVLBTree::insert
  (const int data)
{
  if (getRoot() == NULL)
  {
    setRoot(new LPtrAVLBTreeNode(data));
  }
  else
  {
    setRoot(getRoot()->insert(data));
  }
}


// ---------------------------------------------------------
// Function:       operator=()
// Description:    Copy the entire AVL binary tree.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
LPtrAVLBTree& LPtrAVLBTree::operator =
  (const LPtrAVLBTree& rTree)
{
  if (getRoot() == NULL)
  {
    if (rTree.getRoot() != NULL )
    {
      setRoot(new LPtrAVLBTreeNode(rTree.getRoot()->getSelfData()));
      (LPtrAVLBTreeNode&) *getRoot()= (LPtrAVLBTreeNode&) *rTree.getRoot();
    }
  }
  else
  {
    if (rTree.getRoot() == NULL)
    {
      getRoot()->removeSubtrees();
      delete getRoot();
      setRoot(NULL);
```

```
    }
    else
    {
      (LPtrAVLBTreeNode&) *getRoot()= (LPtrAVLBTreeNode&) *rTree.getRoot();
    }
  }

  return *this;
}
```

Code Listing 9: *LTmplBTreeNode.h*

```
// ************************************************************************
// FILE:        LTmplBTreeNode.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Dec. 12, 1997
// DESCRIPTION:  This file contains declarations of three classes:
//              LTmplBaseBTreeNode, LTmplNullBTreeNode and
//              LTmplFullBTreeNode.
//              Both LTmplNullBTreeNode and LTmplFullBTreeNode are
//              derived from LTmplBaseBTreeNode. The three classes
//              are used to describe a binary tree data structure.
//              The empty node of the bianry tree is not
//              represented by a NULL pointer as in the
//              traditional method, but is represented by a
//              LTmplNullBTreeNode object.
// ************************************************************************

#ifndef LTMPLBTREENODE_H
#define LTMPLBTREENODE_H

#include <iostream.h>
#include <iomanip.h>
#include <fstream.h>
#include <assert.h>

#include "LTypes.h"

// ---------------------------------------------
// Class LTmplBaseBTreeNode defines a set of pure
// virtual funcitons to manipulate the binary
// tree node. Here we acutually view each node as
// a valid binary tree.
// ---------------------------------------------
template <class DATA_TYPE>
class LTmplBaseBTreeNode
{
// -------------------------
// constructors, destructors
// and enum types.
// -------------------------
public:
  enum
  {
    outputWidth_E= 12 // Width used to format the output of the node data
  };

// ------------------------
// Methods
// ------------------------
public:
  // Insert a new data item into the tree. Return the new
  // root for the binary tree.
  virtual LTmplBaseBTreeNode<DATA_TYPE>* insert(const DATA_TYPE data) = 0;

  // Remove a data item from the tree. Return the new root
  // for the binary tree. If the data item is successfully
  // removed, bRemoveOK is set to TRUE, otherwise bRemoveOK
  // is set to FALSE. If this node iteself need to be removed,
  // bDeleteThis is set to TRUE, otherwise it is set to FALSE.
  virtual LTmplBaseBTreeNode<DATA_TYPE>* remove(const DATA_TYPE data, BOOL& bDeleteThis, BOOL& bRemoveOK) = 0;

  // Search a data item in the tree. If find, return TRUE, Otherwise
  // return FALSE.
  virtual BOOL search(const DATA_TYPE data) = 0;

  // Traverse the entire binary tree using the depth first
  // method. Print the result to the standard output.
```

```
    virtual void traverse(void) const = 0;

    // Traverse the entire binary tree using the depth first
    // method. Print the result to a file represented by the
    // ofstream.
    virtual void traverse(ofstream& ofs) const = 0;

    // Return TRUE if this node is a null node. Otherwise
    // return FALSE.
    virtual BOOL isNull(void) const = 0;

    // Return the node data.
    virtual DATA_TYPE getSelfData(void) const = 0;

    // Link the left subtree to a new node.
    virtual void linkLeftSubtree(LTmplBaseBTreeNode<DATA_TYPE>* pLeft) = 0;

    // Link the right subtree to a new node.
    virtual void linkRightSubtree(LTmplBaseBTreeNode<DATA_TYPE>* pRight) = 0;

    // Get the left subtree pointer. The left subtree cannot
    // be modified.
    virtual const LTmplBaseBTreeNode<DATA_TYPE>* getLeftSubtree(void) const = 0;

    // Get the right subtree pointer. The right subtree cannot
    // be modified.
    virtual const LTmplBaseBTreeNode<DATA_TYPE>* getRightSubtree(void) const = 0;

    // Get the left subtree pointer. The left subtree can
    // be modified.
    virtual LTmplBaseBTreeNode<DATA_TYPE>* getLeftSubtree(void) = 0;

    // Get the right subtree pointer. The right subtree can
    // be modified.
    virtual LTmplBaseBTreeNode<DATA_TYPE>* getRightSubtree(void) = 0;

    // Print the node data to the standard output.
    virtual void outputSelfData(void) const = 0;

    // Print the node data to a file.
    virtual void outputSelfData(ofstream& ofs) const = 0;

    // Return the space required for the entire binary tree.
    virtual long space(void) const= 0;
};


// ----------------------------------------------------
// Class LTmplNullBTreeNode implements a null binary tree
// node which has no data and subtrees.
// ----------------------------------------------------
template <class DATA_TYPE>
class LTmplNullBTreeNode: public LTmplBaseBTreeNode<DATA_TYPE>
{

// -------------
// Methods
// -------------
public:

  // Function insert() cannot be inline function
  // as it need to create a LTmplFullBTreeNode. But class
  // LTmplFullBTreeNode is not defined at this point.
  virtual LTmplBaseBTreeNode<DATA_TYPE>* insert(const DATA_TYPE data) ;

// inline methods
  virtual LTmplBaseBTreeNode<DATA_TYPE>* remove(const DATA_TYPE data, BOOL& bDeleteThis, BOOL& bRemoveOK)
  {
    bDeleteThis= FALSE;
    bRemoveOK= FALSE;
```

```
    return this;
  }
  virtual BOOL search(const DATA_TYPE data) {return FALSE;}
  virtual void traverse(void) const {}
  virtual void traverse(ofstream& ofs) const {}
  virtual BOOL isNull(void) const {return TRUE;}
  virtual DATA_TYPE  getSelfData(void) const { cout << "ERROR: cannot get data from a NULL binary tree node." << endl; return
-1;}
  virtual void linkLeftSubtree(LTmplBaseBTreeNode<DATA_TYPE>* pLeft) { cout << "ERROR: cannot set the left subtree for
NULL binary tree node." << endl;}
  virtual void linkRightSubtree (LTmplBaseBTreeNode<DATA_TYPE>* pRight)  { cout << "ERROR: cannot set the right subtree for
NULL binary tree node." << endl;}
  virtual const LTmplBaseBTreeNode<DATA_TYPE>* getLeftSubtree(void) const {cout << "ERROR: cannot get the left tree of a
NULL binary tree node." << endl; return this;}
  virtual const LTmplBaseBTreeNode<DATA_TYPE>* getRightSubtree(void) const { cout << "ERROR: cannot get the right tree of a
NULL binary tree node." << endl; return this;}
  virtual LTmplBaseBTreeNode<DATA_TYPE>* getLeftSubtree(void)  {cout << "ERROR: cannot get the left tree of a NULL binary
tree node." << endl; return this;}
  virtual LTmplBaseBTreeNode<DATA_TYPE>* getRightSubtree(void) { cout << "ERROR: cannot get the right tree of a NULL
binary tree node." << endl; return this;}
  virtual void outputSelfData(void) const { cout << setw(outputWidth_E) << "NIL";}
  virtual void outputSelfData(ofstream& ofs) const { ofs << setw(outputWidth_E) << "NIL";}
  virtual long space(void)  const {return sizeof(*this);}
};


// ----------------------------------------------------
// Class LTmplFullBTreeNode implements a full binary tree
// node which has data and subtrees.
// Since a full binary tree node has pointers to subtree
// nodes, a tree node can be used to represent an entire
// binary tree.
// ----------------------------------------------------
template <class DATA_TYPE>
class LTmplFullBTreeNode: public LTmplBaseBTreeNode<DATA_TYPE>
{

// ------------------------------------------
// Constructors, destructors and enum types.
// ------------------------------------------
public:
  LTmplFullBTreeNode(const DATA_TYPE newData);
  LTmplFullBTreeNode (const LTmplFullBTreeNode<DATA_TYPE> & rNode) {*this= rNode;}
  virtual ~LTmplFullBTreeNode(void);

// ---------
// Methods
// ---------
public:
  virtual LTmplBaseBTreeNode<DATA_TYPE>*   insert   (const DATA_TYPE data);
  virtual LTmplBaseBTreeNode<DATA_TYPE>*   remove   (const DATA_TYPE data, BOOL& bDeleteThis, BOOL& bRemoveOK);

  virtual BOOL  search    (const DATA_TYPE data);
  virtual void  traverse  (void) const;
  virtual void  traverse  (ofstream& ofs) const;
  virtual long  space     (void) const;
  void        removeSubtrees(void);

// operators
  LTmplFullBTreeNode<DATA_TYPE>& operator=(const LTmplFullBTreeNode<DATA_TYPE> &rNode);

// inline methods
  virtual BOOL      isNull        (void) const { return FALSE;}
  virtual DATA_TYPE  getSelfData    (void) const {return m_data;}
  virtual void      linkLeftSubtree  (LTmplBaseBTreeNode<DATA_TYPE>* pLeft) {m_pLeft= pLeft;}
  virtual void      linkRightSubtree  (LTmplBaseBTreeNode<DATA_TYPE>* pRight) {m_pRight= pRight;}
  virtual void      outputSelfData   (void) const { cout << setw(outputWidth_E) << m_data;}
  virtual void      outputSelfData   (ofstream& ofs) const { ofs << setw(outputWidth_E) << m_data;}

  virtual const LTmplBaseBTreeNode<DATA_TYPE>*   getLeftSubtree(void) const { return m_pLeft;}
```

```
    virtual const LTmplBaseBTreeNode<DATA_TYPE>*   getRightSubtree(void) const { return m_pRight;}
    virtual LTmplBaseBTreeNode<DATA_TYPE>*         getLeftSubtree(void) { return m_pLeft;}
    virtual LTmplBaseBTreeNode<DATA_TYPE>*         getRightSubtree(void) { return m_pRight;}

    virtual LTmplNullBTreeNode<DATA_TYPE>*         getStaticNullNode(void) {return &m_nullNode_S;}
    static  LTmplNullBTreeNode<DATA_TYPE>*         getNullNode_S (void) {return &m_nullNode_S;}

  private:
    LTmplFullBTreeNode<DATA_TYPE>* detachSmallestNode(void);

  // ------------
  // Attributes
  // ------------
  private:
    DATA_TYPE                       m_data;     // data kept with the node
    LTmplBaseBTreeNode<DATA_TYPE>     *m_pLeft.   // left child. may point to a LTmplNullBTreeNode or LTmplFullBTreeNode
                            *m_pRight;  // right child.

    static LTmplNullBTreeNode<DATA_TYPE> m_nullNode_S; // used to represent the leaf. It is
                                       // is static since all the leaves can
                                       // use the same null node.
    static BOOL  m_bCountNullNodeSpace_S; // TRUE if the space occupied by the null node is counted.
  };


  ////////////////////////////////////////////////
  ////////////////////////////////////////////////
  // ----------------------------------------------------------
  // Function:       LTmplFullBTreeNode()
  // Descritpion:    Constructor.
  // Errors Generated: None
  // Limitations:    None
  // ----------------------------------------------------------
  template <class DATA_TYPE>
  LTmplFullBTreeNode<DATA_TYPE>::LTmplFullBTreeNode
    (const DATA_TYPE newData)
  {
    m_data= newData;
    m_pLeft= getStaticNullNode();
    m_pRight= getStaticNullNode();
  }

  // ----------------------------------------------------------
  // Function:       ~LTmplFullBTreeNode()
  // Descritpion:    Destructor.
  // Errors Generated: None
  // Limitations:    None
  // ----------------------------------------------------------
  template <class DATA_TYPE>
  LTmplFullBTreeNode<DATA_TYPE>::~LTmplFullBTreeNode
    (void)
  {
  }


  // ----------------------------------------------------------
  // Function:       removeSubtrees()
  // Descritpion:    Remove all the subtrees of the tree node and
  //                 set the left and the right subtree node to be
  //                 null nodes. Memory allocated for the subtrees
  //                 are freed.
  // Errors Generated: None
  // Limitations:
  // ----------------------------------------------------------
  template <class DATA_TYPE>
  void LTmplFullBTreeNode<DATA_TYPE>::removeSubtrees
    (void)
  {
    // ------------------------
    // Delete the left subtree
```

```
  // -------------------------

  if (!m_pLeft->isNull())
  {
    if (m_pLeft->getLeftSubtree()->isNull() && m_pLeft->getRightSubtree()->isNull())
    {
      // m_pLeft is a leaf now. we can delete it.
      delete m_pLeft;
      m_pLeft= getStaticNullNode();
    }
    else
    {
      // Recursively delete the left subtree.
      ((LTmplFullBTreeNode<DATA_TYPE>*) m_pLeft)->removeSubtrees();
    }
  }

  // ---------------------------
  // Delete the right subtree
  // ---------------------------

  if (!m_pRight->isNull())
  {
    if (m_pRight->getLeftSubtree()->isNull() && m_pRight->getRightSubtree()->isNull())
    {
      // m_pRight is a leaf now. we can delete it.
      delete m_pRight;
      m_pRight= getStaticNullNode();
    }
    else
    {
      // Recursively delete the right subtree.
      ((LTmplFullBTreeNode<DATA_TYPE>*) m_pRight)->removeSubtrees();
    }
  }

  // ------------------------------------------------------------
  // By recursively calling removeSubtrees() for the left node and
  // the right node, the two nodes will be turned into leaves.
  // We can delete them now.
  // ------------------------------------------------------------

  if (!m_pLeft->isNull())
  {
    assert(m_pLeft->getLeftSubtree()->isNull());
    assert(m_pLeft->getRightSubtree()->isNull());
    delete m_pLeft;
    m_pLeft= getStaticNullNode();
  }

  if (!m_pRight->isNull())
  {
    assert(m_pRight->getLeftSubtree()->isNull());
    assert(m_pRight->getRightSubtree()->isNull());
    delete m_pRight;
    m_pRight= getStaticNullNode();
  }

  assert(m_pLeft->isNull() && m_pRight->isNull());
  return;

}

// ------------------------------------------------------------
// Function:       insert()
// Description:    Insert a data item into the binary tree.
//                 Return the new root for the binary tree.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
```

```
template <class DATA_TYPE>
LTmplBaseBTreeNode<DATA_TYPE>* LTmplFullBTreeNode<DATA_TYPE>::insert
  (const DATA_TYPE data) // new data item to be inserted
{
  if (data < m_data)
  {
    m_pLeft= m_pLeft->insert(data); // recursive function call
  }
  else if (data > m_data)
  {
    m_pRight= m_pRight->insert(data); // recursive function call
  }
  // else {m_data = data} we don't allow duplication of data

  return this;
}

// ---------------------------------------------------------------
// Function:       remove()
// Descritpion:    Remove a data item from the binary tree.
//                 Return the new root for the binary tree.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
LTmplBaseBTreeNode<DATA_TYPE>* LTmplFullBTreeNode<DATA_TYPE>::remove
  (const DATA_TYPE  data,  // data to be removed
   BOOL&   bDeleteThis, // TRUE if this node itself need to be deleted.
   BOOL&   bRemoveOK)  // TRUE if we sucessfully remove the data item from the tree
{
  bDeleteThis= FALSE;
  bRemoveOK= FALSE;

  if ( data == m_data)
  {
    // ------------------------------------------
    // This node is the node which contains the
    // data item to be removed. Set bDeleteThis to
    // TRUE and bRemoveOK to TRUE.
    // ------------------------------------------
    bDeleteThis= TRUE;
    bRemoveOK= TRUE;

    if (m_pLeft->isNull())
    {
      return m_pRight; // new root is the right subtree.
    }
    else if (m_pRight->isNull())
    {
      return m_pLeft; // new root is the left subtree.
    }
    else
    {
      // ---------------------------------
      // None of the left and right subtree
      // is Null, we should replace the root
      // with the smallest node of the right
      // subtree.
      // ---------------------------------

      LTmplFullBTreeNode<DATA_TYPE>* pNewThisNode= ((LTmplFullBTreeNode*) m_pRight)->detachSmallestNode();

      // Connect the new root with the left subtree.
      pNewThisNode->linkLeftSubtree(m_pLeft);

      // Connect the new root with the right subtree.
      // If the new root is the right subtree itself, then
      // we should not re-link the new root's right subtree.
      if (pNewThisNode != m_pRight)
      {
```

```
        pNewThisNode->linkRightSubtree(m_pRight);
      }

      return pNewThisNode;
    }
  }
  else if ( data < m_data)
  {
    BOOL bDeleteLeft;
    LTmplBaseBTreeNode<DATA_TYPE>* pOldLeft= m_pLeft;

    m_pLeft= m_pLeft->remove(data, bDeleteLeft, bRemoveOK);
    if (bDeleteLeft)
    {
      delete pOldLeft;
    }

    // Data item is removed from the left subtree. The tree
    // root is still this node.
    return this;
  }
  else if (data > m_data)
  {
    BOOL bDeleteRight;
    LTmplBaseBTreeNode<DATA_TYPE>* pOldRight= m_pRight;

    m_pRight= m_pRight->remove(data, bDeleteRight, bRemoveOK);
    if (bDeleteRight)
    {
      delete pOldRight;
    }

    // Data item is removed from the right subtree. The tree
    // root is still this node.
    return this;
  }

  return this;
}


// ----------------------------------------------------------------
// Function:       detachSmallestNode()
// Descritpion:    Detach the smallest (deepest left child) node
//                 from the binary tree and re-connect the
//                 rest of the binary tree. The detached node is
//                 returned.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------------
template <class DATA_TYPE>
LTmplFullBTreeNode<DATA_TYPE>* LTmplFullBTreeNode<DATA_TYPE>::detachSmallestNode
  (void)
{
  if (m_pLeft->isNull())
  {
    // ------------------------------------------------
    // Since all the data items in the right subtree
    // is greater than m_data, the smallest node is
    // this node.
    // ------------------------------------------------
    return this;
  }
  else
  {
    // ------------------------------------------------
    // Find the leftmost node which is the smallest node.
    // ------------------------------------------------

    LTmplFullBTreeNode<DATA_TYPE>* pSmallest= (LTmplFullBTreeNode*) m_pLeft;
```

```
    LTmplFullBTreeNode<DATA_TYPE>* pSmallestParent= this;

    while (!pSmallest->getLeftSubtree()->isNull())
    {
      pSmallestParent= pSmallest;
      pSmallest= (LTmplFullBTreeNode*) pSmallest->getLeftSubtree();
    }

    // ------------------------------------------------
    // Detach the smallest node from the tree. Reconnect
    // the rest of the tree. Return the detached node.
    // ------------------------------------------------

    pSmallestParent->linkLeftSubtree(pSmallest->getRightSubtree());
    return pSmallest;
  }
}

// ---------------------------------------------------------------
// Function:       search()
// Descritpion:    Return TRUE if find data in the tree. Otherwise
//                 return FALSE.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
BOOL LTmplFullBTreeNode<DATA_TYPE>::search
  (const DATA_TYPE data)
{
  BOOL bRet= FALSE;

  if (data == m_data)
  {
    bRet= TRUE;
  }
  else if (data < m_data)
  {
    bRet= m_pLeft->search(data);
  }
  else if (data > m_data)
  {
    bRet= m_pRight->search(data);
  }

  return bRet;
}


// ---------------------------------------------------------------
// Function:       traverse()
// Descritpion:    Output the data in the binary tree to standard
//                 output recursively. Use depth first method.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
void LTmplFullBTreeNode<DATA_TYPE>::traverse
  (void) const
{

  m_pLeft->traverse();

  m_pLeft->outputSelfData();
  cout << setw(outputWidth_E) << "<======";
  cout << setw(outputWidth_E) << m_data;
  cout << setw(outputWidth_E) << "======>";
  m_pRight->outputSelfData();
  cout << endl;

  m_pRight->traverse();
```

```
}

// ---------------------------------------------------------------
// Function:       traverse()
// Descritpion:    Output the data in the binary tree to a file
//                 output stream recursively. Use depth first method.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
void LTmplFullBTreeNode<DATA_TYPE>::traverse
  (ofstream& ofs) const
{
  m_pLeft->traverse(ofs);

  m_pLeft->outputSelfData(ofs);
  ofs << setw(outputWidth_E) << "<======";
  ofs << setw(outputWidth_E) << m_data;
  ofs << setw(outputWidth_E) << "======>";
  m_pRight->outputSelfData(ofs);
  ofs << endl;

  m_pRight->traverse(ofs);
}
// ---------------------------------------------------------------
// Function:       operator =()
// Descritpion:    Copy the entire binary tree, including the
//                 subtrees. New memory is allocated.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
LTmplFullBTreeNode<DATA_TYPE>& LTmplFullBTreeNode<DATA_TYPE>::operator =
  (const LTmplFullBTreeNode<DATA_TYPE> &rNode)
{

  static int flag= 0;
  // ---------------------------
  // Delete the existing tree.
  // ---------------------------

  if (flag == 0)
  {
    removeSubtrees();
    flag= 1;
    m_data= rNode.getSelfData();
  }

  if (!rNode.getLeftSubtree()->isNull())
  {
    m_pLeft= new LTmplFullBTreeNode(rNode.getLeftSubtree()->getSelfData());
    *((LTmplFullBTreeNode<DATA_TYPE>*) m_pLeft)= *((LTmplFullBTreeNode<DATA_TYPE>*)rNode.getLeftSubtree());
  }

  if (!rNode.getRightSubtree()->isNull())
  {
    m_pRight= new LTmplFullBTreeNode(rNode.getRightSubtree()->getSelfData());
    *((LTmplFullBTreeNode<DATA_TYPE>*) m_pRight)= *((LTmplFullBTreeNode<DATA_TYPE>*)rNode.getRightSubtree());
  }

  flag= 0;

  return *this;
}

// ---------------------------------------------------------------
// Function:       space()
// Descritpion:    Return the space of the entire binary tree.
//                 Note that in an application, all the empty
//                 nodes in all the binary trees are actually
```

```
//              represented by the same static null node.
//              Therefore we only calculate the space
//              occupied by the null node once in the
//              application.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
long LTmplFullBTreeNode<DATA_TYPE>::space
  (void) const
{
  long treeSpace= sizeof(*this);
  if (!m_pLeft->isNull())
  {
    treeSpace+= m_pLeft->space();
  }

  if (!m_pRight->isNull())
  {
    treeSpace+= m_pRight->space();
  }

  if (!m_bCountNullNodeSpace_S)
  {
    treeSpace+= m_nullNode_S.space();
    m_bCountNullNodeSpace_S= TRUE;
  }
  return treeSpace;
}


////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
// Implementation for class LTmplNullBTreeNode
//

// ---------------------------------------------------------------
// Function:       insert
// Descritpion:    Insert a data item into the null node.
//                 Return a full node.
// Errors Generated: None
// Limitations:    Didn't handle the memory allocation failure
//                 if occurred.
// ---------------------------------------------------------------
template <class DATA_TYPE>
LTmplBaseBTreeNode<DATA_TYPE>* LTmplNullBTreeNode<DATA_TYPE>::insert
  (const DATA_TYPE data)
{
  return new LTmplFullBTreeNode<DATA_TYPE>(data);
}


#endif //LTMPLBTREENODE_H
```

Code Listing 10: *LTmplBTree.h*

```
// ****************************************************************
// FILE:        LTmplBTree.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Dec. 31, 1997
// DESCRIPTION:  LTmplBTree encapsulates a binary tree data structure.
//               It uses a LTmplBaseBTreeNode object as its root.
// ****************************************************************

#ifndef LTMPLBTREE_H
#define LTMPLBTREE_H

#include <fstream.h>
#include <assert.h>

#include "LTypes.h"
#include "LTmplBTreeNode.h"

template <class DATA_TYPE>
class LTmplBTree
{
// ----------------------------
// Constructors, destructors and
// enum types
// ----------------------------

public:
  LTmplBTree(void);
  virtual ~LTmplBTree(void);

// -------------------------
// Member functions
// -------------------------
public:
  BOOL remove (const DATA_TYPE data);
  void empty(void);

// operators
  LTmplBTree<DATA_TYPE>& operator =(const LTmplBTree<DATA_TYPE>& tree);

// inline functions
  void insert   (const DATA_TYPE data) {m_pRoot= m_pRoot->insert(data);}
  void traverse  (void) const {m_pRoot->traverse();}
  void traverse  (ofstream& ofs) const {m_pRoot->traverse(ofs);}
  BOOL search   (const DATA_TYPE data) const {return m_pRoot->search(data);}
  long space    (void)  const { return (sizeof(*this) + m_pRoot->space());}

  const LTmplBaseBTreeNode<DATA_TYPE>&   getRoot(void) const {return *m_pRoot;}

// ----------------------
// Attributes
// ----------------------
private:
  LTmplBaseBTreeNode<DATA_TYPE>* m_pRoot;
};


////////////////////////////////
////////////////////////////////

// ---------------------------------------------------------
// Function:       LTmplBTree()
// Descritpion:    Default constructor. Initialize the root
//                 to be the static LNullBTreeNode member
//                 variable defined in LTmplFullBTreeNode class.
//                 class.
// Errors Generated: None
```

```
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplBTree<DATA_TYPE>::LTmplBTree
  (void)
{
  m_pRoot= LTmplFullBTreeNode<DATA_TYPE>::getNullNode_S();
}

// ---------------------------------------------------------
// Function:       ~LTmplBTree()
// Descripion:     Destructor. Delete the entire binary tree.
//                 Free memory allocated.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplBTree<DATA_TYPE>::~LTmplBTree(void)
{
  empty();
}

// ---------------------------------------------------------
// Function:       remove()
// Descripion:     Remove data from the binary tree. Return
//                 TRUE if found the data and successfully
//                 remove it. Otherwise return FALSE.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
BOOL LTmplBTree<DATA_TYPE>::remove
  (const DATA_TYPE data)
{
  BOOL bDeleteRoot,
     bRemoveOK;

  LTmplBaseBTreeNode<DATA_TYPE>* pOldRoot= m_pRoot;

  m_pRoot= m_pRoot->remove(data, bDeleteRoot, bRemoveOK);

  if (bDeleteRoot)
  {
    delete pOldRoot;
  }

  return bRemoveOK;
}

// ---------------------------------------------------------
// Function:       operator = ()
// Descripion:     Copy the entire binary tree. Allocate
//                 memory as needed.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplBTree<DATA_TYPE>& LTmplBTree<DATA_TYPE>::operator =
  (const LTmplBTree<DATA_TYPE>& tree)
{
  if (m_pRoot->isNull())
  {
    if (!tree.getRoot().isNull())
    {
      m_pRoot= new LTmplFullBTreeNode<DATA_TYPE>(tree.getRoot().getSelfData());
      *((LTmplFullBTreeNode<DATA_TYPE>*)m_pRoot)= (LTmplFullBTreeNode<DATA_TYPE>&)tree.getRoot();
    }
  }
  else
  {
```

```
    if (tree.getRoot().isNull())
    {
      ((LTmplFullBTreeNode<DATA_TYPE>*) m_pRoot)->removeSubtrees();
      delete m_pRoot;

      m_pRoot= LTmplFullBTreeNode<DATA_TYPE>::getNullNode_S();
    }
    else
    {
      *((LTmplFullBTreeNode<DATA_TYPE>*) m_pRoot)= (LTmplFullBTreeNode<DATA_TYPE>& )tree.getRoot();
    }
  }

  return *this;
}

// ------------------------------------------------------------
// Function:       empty()
// Description:    Empty the entire tree. Set the root to
//                 be a null tree node.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
template <class DATA_TYPE>
void LTmplBTree<DATA_TYPE>::empty
  (void)
{
  if (!m_pRoot->isNull())
  {
    ((LTmplFullBTreeNode<DATA_TYPE>*) m_pRoot)->removeSubtrees();
    delete m_pRoot;

    m_pRoot= LTmplFullBTreeNode<DATA_TYPE>::getNullNode_S();
  }
}


#endif // LTMPLBTREE_H
```

Code Listing 11: *LTmplPtrBTreeNode.h*

```
// ****************************************************************
// FILE:        LTmplPtrBTreeNode.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Feb. 20, 1998
// DESCRIPTION:  Class LTmplPtrBTreeNode implements a template
//              binary tree node. The node itself actually
//              represents a valid binary tree. The empty
//              nodes of the tree are represented using
//              NULL pointers.
// ****************************************************************

#ifndef LTMPLPTRBTREENODE_H
#define LTMPLPTRBTREENODE_H

#include <fstream.h>
#include "LTypes.h"
#include <assert.h>
#include <iostream.h>
#include <iomanip.h>


template <class DATA_TYPE>
class LTmplPtrBTreeNode
{
// ------------------------------------------
// Constructors, destructors and enum types.
// ------------------------------------------
public:
  LTmplPtrBTreeNode(const DATA_TYPE data);
  LTmplPtrBTreeNode(const LTmplPtrBTreeNode<DATA_TYPE>& rNode);
  virtual ~LTmplPtrBTreeNode(void);

  enum
  {
    outputWidth_E= 12
  };

// ------------------
// Methods
// ------------------
public:
  virtual LTmplPtrBTreeNode<DATA_TYPE>* insert(const DATA_TYPE data);
  virtual LTmplPtrBTreeNode<DATA_TYPE>* remove(const DATA_TYPE data, BOOL& bDeleteThis, BOOL& bRemoveOK);
  virtual void traverse(void) const;
  virtual void traverse(ofstream& ofs) const;
  virtual void removeSubtrees(void);

  virtual long space(void) const;
  BOOL search(const DATA_TYPE data);

// operators
  LTmplPtrBTreeNode<DATA_TYPE>& operator = (const LTmplPtrBTreeNode<DATA_TYPE>& rNode);

// inline methods
  void          linkLeftSubtree   (LTmplPtrBTreeNode<DATA_TYPE>* pLeft) {m_pLeft= pLeft;}
  void          linkRightSubtree  (LTmplPtrBTreeNode<DATA_TYPE>* pRight) {m_pRight= pRight;}
  DATA_TYPE     getSelfData       (void) const {return m_data;}
  void          setSelfData       (const DATA_TYPE data) {m_data= data;}

  LTmplPtrBTreeNode<DATA_TYPE>*       getLeftSubtree   (void) {return m_pLeft;}
  LTmplPtrBTreeNode<DATA_TYPE>*       getRightSubtree  (void) {return m_pRight;}
  const LTmplPtrBTreeNode<DATA_TYPE>* getLeftSubtree   (void) const {return m_pLeft;}
  const LTmplPtrBTreeNode<DATA_TYPE>* getRightSubtree  (void) const {return m_pRight;}

private:
  LTmplPtrBTreeNode<DATA_TYPE>* detachSmallestNode(void);
```

```
// ---------------
// Attributes
// ---------------
private:
  DATA_TYPE                m_data;
  LTmplPtrBTreeNode<DATA_TYPE>* m_pLeft;
  LTmplPtrBTreeNode<DATA_TYPE>* m_pRight;

};



////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////
// Implementation


// ---------------------------------------------------------
// Function:       LTmplPtrBTreeNode()
// Descritpion:    Constructor.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTreeNode<DATA_TYPE>::LTmplPtrBTreeNode
  (const DATA_TYPE newData)
{
  m_data= newData;
  m_pLeft= NULL;
  m_pRight= NULL;
}

// ---------------------------------------------------------
// Function:       LTmplPtrBTreeNode()
// Descritpion:    Copy constructor.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTreeNode<DATA_TYPE>::LTmplPtrBTreeNode
  (const LTmplPtrBTreeNode& rNode)
{
  *this= rNode;
}

// ---------------------------------------------------------
// Function:       ~LTmplPtrBTreeNode()
// Descritpion:    Destructor.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTreeNode<DATA_TYPE>::~LTmplPtrBTreeNode
  (void)
{
}


// ---------------------------------------------------------
// Function:       insert()
// Descritpion:    Insert a data item into the binary tree.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTreeNode<DATA_TYPE>* LTmplPtrBTreeNode<DATA_TYPE>::insert
  (const DATA_TYPE data)
{
else if (data < m_data)
```

```
  {
    if (m_pLeft != NULL)
    {
      m_pLeft->insert(data);
    }
    else
    {
      m_pLeft= new LTmplPtrBTreeNode<DATA_TYPE> (data);
    }
  }
  else if (data > m_data)
  {
    if (m_pRight != NULL)
    {
      m_pRight->insert(data);
    }
    else
    {
      m_pRight= new LTmplPtrBTreeNode<DATA_TYPE> (data);
    }
  }

  return this;
}


// ------------------------------------------------------------
// Function:        remove()
// Descritpion:     Remove a data item from the binary tree.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTreeNode<DATA_TYPE>* LTmplPtrBTreeNode<DATA_TYPE>::remove
  (const DATA_TYPE data,    // data to be removed
   BOOL& bDeleteThis,  // TRUE if this node itself need to be deleted
   BOOL& bRemoveOK)    // TRUE if the data item is successfully removed from the binary tree.
{
  bDeleteThis= FALSE;
  bRemoveOK= FALSE;

  if (data == m_data)
  {
    bDeleteThis= TRUE;
    bRemoveOK= TRUE;
    if (m_pLeft == NULL)
    {
      return m_pRight;
    }
    else if (m_pRight == NULL)
    {
      return m_pLeft;
    }
    else
    {
      LTmplPtrBTreeNode<DATA_TYPE>* pNewThisNode= m_pRight->detachSmallestNode();

      pNewThisNode->linkLeftSubtree(m_pLeft);

      if (pNewThisNode != m_pRight)
      {
        pNewThisNode->linkRightSubtree(m_pRight);
      }

      return pNewThisNode;
    }
  }
  else if ( data < m_data)
  {
    if (m_pLeft == NULL)
```

```
    {
      // We cannot find the data item. bRemoveOK
      // and bDeleteThis are set to FALSE already
      // Root is still this node, so return this.
      return this;
    }

    BOOL bDeleteLeft;
    LTmplPtrBTreeNode<DATA_TYPE>* pOldLeft= m_pLeft;

    m_pLeft= m_pLeft->remove(data, bDeleteLeft, bRemoveOK);
    if (bDeleteLeft)
    {
      delete pOldLeft;
    }
    return this;
  }
  else if (data > m_data)
  {
    if (m_pRight == NULL)
    {
      // We cannot find the data item. bRemoveOK
      // and bDeleteThis are set to FALSE alreay.
      // Root is still this node, so return this.
      return this;
    }
    BOOL bDeleteRight;
    LTmplPtrBTreeNode<DATA_TYPE>* pOldRight= m_pRight;

    m_pRight= m_pRight->remove(data, bDeleteRight, bRemoveOK);
    if (bDeleteRight)
    {
      delete pOldRight;
    }
    return this;
  }

  return this;
}

// ------------------------------------------------------------------
// Function:       search()
// Descritpion:    Search data in the binary tree. If found,
//                 return TRUE, otherwise return FALSE.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------------
template <class DATA_TYPE>
BOOL LTmplPtrBTreeNode<DATA_TYPE>::search
  (const DATA_TYPE data)
{
  BOOL bRet= FALSE;

  if (data == m_data)
  {
    bRet= TRUE;
  }
  else if (data < m_data)
  {
    if (m_pLeft != NULL)
      bRet= m_pLeft->search(data);
  }
  else if (data > m_data)
  {
    if (m_pRight != NULL)
      bRet= m_pRight->search(data);
  }

  return bRet;
}
```

```
// ---------------------------------------------------------------
// Function:       traverse()
// Descritpion:    Output the data in the binary tree to the
//                 standard output recursively.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
void LTmplPtrBTreeNode<DATA_TYPE>::traverse
  (void) const
{
  if (m_pLeft != NULL)
  {
    m_pLeft->traverse();
    cout << setw(outputWidth_E) << m_pLeft->getSelfData();
  }
  else
  {
    cout << setw(outputWidth_E) << "NIL";
  }

  cout << setw(outputWidth_E) << "<======";
  cout << setw(outputWidth_E) << m_data;
  cout << setw(outputWidth_E) << "======>";

  if (m_pRight != NULL)
  {
    cout << setw(outputWidth_E) << m_pRight->getSelfData() << endl;
    m_pRight->traverse();
  }
  else
  {
    cout << setw(outputWidth_E) << "NIL" << endl;
  }

}


// ---------------------------------------------------------------
// Function:       traverse()
// Descritpion:    Output the data in the binary tree to a
//                 file output stream recursively.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
void LTmplPtrBTreeNode<DATA_TYPE>::traverse
  (ofstream& ofs) const
{
  if (m_pLeft != NULL)
  {
    m_pLeft->traverse(ofs);
    ofs << setw(outputWidth_E) << m_pLeft->getSelfData();
  }
  else
  {
    ofs << setw(outputWidth_E) << "NIL";
  }

  ofs << setw(outputWidth_E) << "<======";
  ofs << setw(outputWidth_E) << m_data;
  ofs << setw(outputWidth_E) << "======>";

  if (m_pRight != NULL)
  {
    ofs << setw(outputWidth_E) << m_pRight->getSelfData() << endl;
    m_pRight->traverse(ofs);
  }
  else
```

```
  {
    ofs << setw(outputWidth_E) << "NIL" << endl;
  }

}

// ---------------------------------------------------------------
// Function:       operator=()
// Description:    Copy the entire binary tree, including the
//                 subtrees. New memory is allocated to copy
//                 the entire tree structure.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTreeNode<DATA_TYPE>& LTmplPtrBTreeNode<DATA_TYPE>::operator =
  (const LTmplPtrBTreeNode<DATA_TYPE>& rNode)
{

  static int flag= 0;

  // ---------------------------
  // Delete the existing tree.
  // ---------------------------

  if (flag == 0)
  {
    removeSubtrees();
    flag ++;
    m_data= rNode.getSelfData();
  }

  if (rNode.getLeftSubtree() != NULL)
  {
    m_pLeft= new LTmplPtrBTreeNode<DATA_TYPE> (rNode.getLeftSubtree()->getSelfData());
    *m_pLeft= *rNode.getLeftSubtree();
  }

  if (rNode.getRightSubtree() != NULL)
  {
    m_pRight= new LTmplPtrBTreeNode<DATA_TYPE> (rNode.getRightSubtree()->getSelfData());
    *m_pRight= *rNode.getRightSubtree();
  }

  flag= 0;

  return *this;

}


// ---------------------------------------------------------------
// Function:       removeSubtrees()
// Description:    Remove all the subtrees of the tree node and
//                 set the left and the right subtrees to be NULL.
//                 Free the memory allocated for the subtrees.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class DATA_TYPE>
void LTmplPtrBTreeNode<DATA_TYPE>::removeSubtrees
  (void)
{
  // ------------------------
  // Delete the left subtree
  // ------------------------

  if (m_pLeft != NULL)
  {
    if ((m_pLeft->getLeftSubtree() == NULL) &&
```

```
      (m_pLeft->getRightSubtree() == NULL))
    {
      // m_pLeft is a leaf, we can delete it.
      delete m_pLeft;
      m_pLeft= NULL;
    }
    else
    {
      m_pLeft->removeSubtrees();
    }
  }

  // ---------------------------
  // Delete the right subtree
  // ---------------------------

  if (m_pRight != NULL)
  {
    if ((m_pRight->getLeftSubtree() == NULL) &&
        (m_pRight->getRightSubtree() == NULL))
    {
      // m_pRight is a leaf, we can delete it.
      delete m_pRight;
      m_pRight= NULL;
    }
    else
    {
      m_pRight->removeSubtrees();
    }
  }

  // ------------------------------------------------------------
  // By recursively calling removeSubtrees() for the left node and
  // the right node, the two nodes become leaves, we can delete
  // them.
  // ------------------------------------------------------------

  if (m_pLeft != NULL)
  {
    assert(m_pLeft->getLeftSubtree() == NULL);
    assert(m_pLeft->getRightSubtree() == NULL);
    delete m_pLeft;
    m_pLeft= NULL;
  }

  if (m_pRight != NULL)
  {
    assert(m_pRight->getLeftSubtree() == NULL);
    assert(m_pRight->getRightSubtree() == NULL);
    delete m_pRight;
    m_pRight= NULL;
  }

  assert((m_pLeft == NULL) && (m_pRight == NULL));

  return;

}

// ------------------------------------------------------------
// Function:        detachSmallestNode()
// Descritpion:     Detach the smallest (deepest left child) node
//                  from the binary tree and re-connect the
//                  rest of the binary tree. Return the detached
//                  smallest node.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTreeNode<DATA_TYPE>* LTmplPtrBTreeNode<DATA_TYPE>::detachSmallestNode
```

```
  (void)
{
  if (m_pLeft == NULL)
  {
    return this;
  }
  else
  {
    LTmplPtrBTreeNode<DATA_TYPE>* pSmallest= m_pLeft;
    LTmplPtrBTreeNode<DATA_TYPE>* pSmallestParent= this;

    while (pSmallest->getLeftSubtree() != NULL)
    {
      pSmallestParent= pSmallest;
      pSmallest= pSmallest->getLeftSubtree();
    }

    pSmallestParent->linkLeftSubtree(pSmallest->getRightSubtree());
    return pSmallest;
  }
}


// ------------------------------------------------------------------
// Function:       space()
// Descritpion:    Return the space needed for the entire tree.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------------
template <class DATA_TYPE>
long LTmplPtrBTreeNode<DATA_TYPE>::space
  (void) const
{

  long treeSpace= sizeof(*this);

  if (m_pLeft != NULL)
  {
    treeSpace+= m_pLeft->space();
  }

  if (m_pRight != NULL)
  {
    treeSpace+= m_pRight->space();
  }

  return treeSpace;
}

#endif // LTMPLPTRBTREENODE_H
```

Code Listing 12: *LTmplPtrBTree.h*

```
// ****************************************************************
// FILE:        LTmplPtrBTree.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Feb. 20. 1998
// DESCRIPTION:  LTmplPtrBTree encapsulates a template binary
//              tree data structure.
//              It uses a LTmplPtrBTreeNode pointer as its root.
// ****************************************************************

#ifndef LTMPLPTRBTREE_H
#define LTMPLPTRBTREE_H

#include <stddef.h>
#include <assert.h>

#include "LTypes.h"
#include "LTmplPtrBTreeNode.h"


template <class DATA_TYPE>
class LTmplPtrBTree
{
// ----------------------------
// Constructors, destructors and
// enum types
// ----------------------------

public:

  LTmplPtrBTree(void);
  LTmplPtrBTree(const LTmplPtrBTree<DATA_TYPE>& rTree) {*this = rTree;}
  virtual ~LTmplPtrBTree(void);

// ------------------------
// Member functions
// ------------------------
public:
  virtual void  insert    (const DATA_TYPE data);
  BOOL          remove    (const DATA_TYPE data);
  void          empty     (void);
  BOOL          search    (const DATA_TYPE data) const;

// operators
  LTmplPtrBTree<DATA_TYPE>& operator = (const LTmplPtrBTree<DATA_TYPE>& rTree);

// inline functions
  void traverse   (void) const { if (m_pRoot != NULL) m_pRoot->traverse();}
  void traverse   (ofstream& ofs) const { if (m_pRoot != NULL) m_pRoot->traverse(ofs);}
  long space      (void) const { if (m_pRoot != NULL) return (sizeof(*this) + m_pRoot->space()); else return sizeof(*this);}

  const LTmplPtrBTreeNode<DATA_TYPE>* getRoot  (void) const {return m_pRoot;}
  LTmplPtrBTreeNode<DATA_TYPE>*       getRoot  (void)      {return m_pRoot;}
  void                                setRoot  (LTmplPtrBTreeNode<DATA_TYPE>* pRoot) {m_pRoot= pRoot;}

// ----------------------
// Attributes
// ----------------------
private:
  LTmplPtrBTreeNode<DATA_TYPE>* m_pRoot;

};


////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
```

```
// Implementation
//

// ---------------------------------------------------------
// Function:        LTmplPtrBTree()
// Descritpion:     Default constructor.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTree<DATA_TYPE>::LTmplPtrBTree
  (void)
{
  m_pRoot= NULL;
}


// ---------------------------------------------------------
// Function:        ~LTmplPtrBTree()
// Descritpion:     Destructor. Delete the root node.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTree<DATA_TYPE>::~LTmplPtrBTree(void)
{
  empty();
}

// ---------------------------------------------------------
// Function:        insert()
// Descritpion:     insert new data into the binary tree.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
void LTmplPtrBTree<DATA_TYPE>::insert
  (const DATA_TYPE data)
{
  if (m_pRoot == NULL)
  {
    m_pRoot= new LTmplPtrBTreeNode<DATA_TYPE> (data);
  }
  else
  {
    m_pRoot= m_pRoot->insert(data);
  }
}

// ---------------------------------------------------------
// Function:        remove()
// Descritpion:     Remove data from the binary tree. Return
//                  TRUE if found the data and successfully
//                  remove it. Otherwise return FALSE.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
BOOL LTmplPtrBTree<DATA_TYPE>::remove
  (const DATA_TYPE data)
{
  if (m_pRoot == NULL)
    return FALSE;

  BOOL bDeleteRoot,
     bRemoveOK;

  LTmplPtrBTreeNode<DATA_TYPE>* pOldRoot= m_pRoot;

  m_pRoot= m_pRoot->remove(data, bDeleteRoot, bRemoveOK);
```

```
    if (bDeleteRoot)
    {
      delete pOldRoot;
    }

    return bRemoveOK;
}

// ---------------------------------------------------------
// Function:        operator=()
// Description:     Copy the entire binary tree.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrBTree<DATA_TYPE>& LTmplPtrBTree<DATA_TYPE>::operator =
  (const LTmplPtrBTree<DATA_TYPE>& rTree)
{
  if (m_pRoot == NULL)
  {
    if (rTree.getRoot() != NULL )
    {
      m_pRoot= new LTmplPtrBTreeNode<DATA_TYPE> (rTree.getRoot()->getSelfData());
      *m_pRoot= *rTree.getRoot();
    }
  }
  else
  {
    if (rTree.getRoot() == NULL)
    {
      m_pRoot->removeSubtrees();
      delete m_pRoot;
      m_pRoot= NULL;
    }
    else
    {
      *m_pRoot= *rTree.getRoot();
    }
  }

  return *this;
}

// ---------------------------------------------------------
// Function:        empty()
// Description:     Empty the entire tree. Set the root to
//                  be a NULL pointer
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
void LTmplPtrBTree<DATA_TYPE>::empty
  (void)
{
  if (m_pRoot != NULL)
  {
    m_pRoot->removeSubtrees();
    delete m_pRoot;
    m_pRoot= NULL;
  }
}

// ---------------------------------------------------------
// Function:        search()
// Description:     Check if data is contained in the binary
//                  tree. If yes, return TRUE. Otherwise return
//                  FALSE.
// Errors Generated: None
// Limitations:
```

```
// ------------------------------------------------------------
template <class DATA_TYPE>
BOOL LTmplPtrBTree<DATA_TYPE>::search
  (const DATA_TYPE data) const
{
  if (m_pRoot == NULL)
    return FALSE;
  else
    return m_pRoot->search(data);
}


#endif // LTMPLPTRBTREE_H
```

Code Listing 13: *LAVLBTreeNode.h*

```
// ****************************************************************
// FILE:        LAVLBTreeNode.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Jan. 20, 1998
// DESCRIPTION:  This file contains declarations of two classes:
//              LNullAVLBTreeNode and LFullAVLBTreeNode.
// ****************************************************************

#ifndef LAVLBTREENODE_H
#define LAVLBTREENODE_H

#include "LBTreeNode.h"

// ------------------------------------------------
// Class LNullAVLBTreeNode implements a null node for
// an AVL binary tree.
// ------------------------------------------------
class LNullAVLBTreeNode: public LNullBTreeNode
{

// -------------
// Methods
// -------------
public:

  // Function insert() cannot be inline function
  // as it need to create a LFullAVLBTreeNode. But class
  // LFullAVLBTreeNode is not defined at this point.
  virtual LBaseBTreeNode* insert(const int data) ;

// inline methods
  virtual int getHeight(void) const {return -1;}
  virtual long space (void) const { return sizeof(*this);}
};


// ------------------------------------------------
// Class LFullAVLBTreeNode implements a full binary tree
// node  for an AVL tree. It is subclassed from
// LFullBTreeNode. Each node will keep its height
// information and is always balanced.
// ------------------------------------------------
class LFullAVLBTreeNode: public LFullBTreeNode
{

// ------------------------------------------
// Constructors, destructors and enum types.
// ------------------------------------------
public:
  LFullAVLBTreeNode(const int newData);
  LFullAVLBTreeNode(const LFullAVLBTreeNode& rNode) : LFullBTreeNode(rNode.getSelfData()) {*this= rNode;}

// ---------
// Methods
// ---------
public:
  virtual LBaseBTreeNode* insert(const int data) ;
  virtual LBaseBTreeNode* remove(const int data, BOOL& bDeleteThis, BOOL& bRemoveOK);

// operators
  LFullAVLBTreeNode& operator= (const LFullAVLBTreeNode& rNode);

  // inlines
  virtual LNullBTreeNode*  getStaticNullNode(void) {return &m_nullAVLNode_S;}
  static  LNullBTreeNode*  getNullNode_S (void) {return &m_nullAVLNode_S;}
```

```
  virtual int  getHeight(void) const {return m_height;}
  virtual void setHeight(const int height) { m_height= height;}
  virtual long space (void) const;

public:
  void            calculateHeight    (void);
  LFullAVLBTreeNode* balance          (const char direction);

private:
  LFullAVLBTreeNode* detachSmallestNode  (void);
  LFullAVLBTreeNode* singleRotateLeft    (void);
  LFullAVLBTreeNode* doubleRotateLeft    (void);
  LFullAVLBTreeNode* singleRotateRight   (void);
  LFullAVLBTreeNode* doubleRotateRight   (void);

// ------------
// Attributes
// ------------
private:
  int                m_height;
  static LNullAVLBTreeNode m_nullAVLNode_S; // used to represent the leaf. It is
                                            // static since all the leaves can
                                            // use the same null node.
  static BOOL   m_bCountNullAVLNodeSpace_S;
};

#endif //LAVLBTREENODE_H
```

Code Listing 14: *LAVLBTreeNode.cpp*

```
// ****************************************************************
// FILE:        LAVLBTreeNode.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Jan. 20, 1998
// DESCRIPTION:  Refer to LAVLBTreeNode.h
// ****************************************************************

#define LAVLBTREENODE_CPP
// -------------
// Includes
// -------------

#include <iostream.h>
#include <assert.h>

#include "LAVLBTreeNode.h"
#include "LTmplStack.h"

#define MAX(a, b) ( ((a) >= (b))? (a) : (b) )


// ------------------------------------------------
// Initialization of the static member variables.
// ------------------------------------------------

LNullAVLBTreeNode LFullAVLBTreeNode::m_nullAVLNode_S;
BOOL          LFullAVLBTreeNode::m_bCountNullAVLNodeSpace_S= FALSE;
LTmplNullStackNode<LFullAVLBTreeNode*> LTmplFullStackNode<LFullAVLBTreeNode*> :: m_nullNode_S;


// ------------------------------------------------------------
// Function:       LFullAVLBTreeNode()
// Descritpion:    Constructor.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
LFullAVLBTreeNode::LFullAVLBTreeNode
  (const int newData)
  :LFullBTreeNode(newData),
   m_height(0)
{
  linkLeftSubtree(getStaticNullNode());
  linkRightSubtree(getStaticNullNode());

  return;
}

// ------------------------------------------------------------
// Function:       insert()
// Descritpion:    Insert a data item into the binary tree.
//                 Return the new root for the binary tree.
//                 Re-balance the tree when needed.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
LBaseBTreeNode* LFullAVLBTreeNode::insert
  (const int data) // new data item to be inserted
{
  LFullAVLBTreeNode* pNewRoot= this;

  if ( data < getSelfData() )
  {
    linkLeftSubtree(getLeftSubtree()->insert(data));
    pNewRoot= balance('L');
  }
  else if ( data > getSelfData() )
```

```
  {
    linkRightSubtree(getRightSubtree()->insert(data));
    pNewRoot= balance('R');
  }
  // else m_data == getSelfData(). Don't do anything.

  return pNewRoot;
}

// ------------------------------------------------------------
// Function:       remove()
// Descritpion:  Remove a data item from the binary tree.
//                    Return the new root for the binary tree.
//                    Re-balance the tree when needed.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
LBaseBTreeNode* LFullAVLBTreeNode::remove
  (const int  data,       // data to be removed
   BOOL&     bDeleteThis,  // TRUE if this node itself need to be deleted.
   BOOL&     bRemoveOK)    // TRUE if we sucessfully remove the data item from the tree
{

  LBaseBTreeNode* pNewRoot;

  bDeleteThis= FALSE;
  bRemoveOK= FALSE;

  if ( data == getSelfData())
  {
    // ---------------------------------------
    // This node is the node which contains the
    // data item to be removed. Set bDeleteThis to
    // TRUE and bRemoveOK to TRUE.
    // ---------------------------------------
    bDeleteThis= TRUE;
    bRemoveOK= TRUE;

    if (getLeftSubtree()->isNull())
    {
      pNewRoot= getRightSubtree(); // new root is the right subtree.
    }
    else if (getRightSubtree()->isNull())
    {
      pNewRoot= getLeftSubtree(); // new root is the left subtree.
    }
    else
    {
      // ---------------------------------
      // None of the left and right subtree
      // is Null. we should replace the root
      // with the smallest node of the right
      // subtree.
      // ---------------------------------

      LFullAVLBTreeNode* pNewThisNode;

      pNewThisNode= ((LFullAVLBTreeNode* ) getRightSubtree())->detachSmallestNode();

      // Connect the new root with the left subtree.
      pNewThisNode->linkLeftSubtree(getLeftSubtree());
      pNewThisNode->calculateHeight();

      // ---------------------------------
      // Re-balance the new tree.
      // ---------------------------------

      pNewThisNode= pNewThisNode->balance('L');
      pNewThisNode->calculateHeight();
      pNewRoot= pNewThisNode;
```

```
    }
  }
  else if ( data < getSelfData())
  {
    BOOL bDeleteLeft;
    LBaseBTreeNode* pOldLeft= getLeftSubtree();

    linkLeftSubtree(getLeftSubtree()->remove(data, bDeleteLeft, bRemoveOK));

    if (bDeleteLeft)
    {
      delete pOldLeft;
    }

    // ---------------------------------------------------------
    // Data item is removed from the left subtree. The tree
    // root is still this node.
    // ---------------------------------------------------------

    pNewRoot= this;

    // ---------------------------------------------------------
    // If the data is successfully deleted, and the tree is
    // not balanced yet, we should re-balance the tree.
    // Note that deleting a node from the left sub-tree is
    // equivalent to inserting a node in the right sub-tree.
    // ---------------------------------------------------------

    if (bRemoveOK )
    {
      pNewRoot= balance('R');
    }
  }
  else if (data > getSelfData())
  {
    BOOL bDeleteRight;
    LBaseBTreeNode* pOldRight= getRightSubtree();

    linkRightSubtree(getRightSubtree()->remove(data, bDeleteRight, bRemoveOK));
    if (bDeleteRight)
    {
      delete pOldRight;
    }

    // ---------------------------------------------------------
    // Data item is removed from the right subtree. The tree
    // root is still this node.
    // ---------------------------------------------------------

    pNewRoot= this;

    // ---------------------------------------------------------
    // If the data is successfully deleted, and the tree is
    // not balanced yet, we should re-balance the tree.
    // Note that deleting a node from the right sub-tree is
    // equivalent to inserting a node in the left sub-tree.
    // ---------------------------------------------------------

    if (bRemoveOK)
    {
      pNewRoot= balance('L');
    }
  }
  return pNewRoot;
}


// ---------------------------------------------------------------
// Function:      operator= ()
```

```
// Descritpion:     Copy the entire binary tree. including the
//                  subtrees. New memory is allocated.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
LFullAVLBTreeNode& LFullAVLBTreeNode::operator=
  (const LFullAVLBTreeNode& rNode)
{

  static int flag= 0;
  // ---------------------------
  // Delete the existing tree.
  // ---------------------------

  if (flag == 0)
  {
    removeSubtrees();
    flag= 1;
    setSelfData(rNode.getSelfData());
    m_height= rNode.getHeight();
  }

  if (!rNode.getLeftSubtree()->isNull())
  {
    linkLeftSubtree(new LFullAVLBTreeNode(rNode.getLeftSubtree()->getSelfData()));
    *((LFullAVLBTreeNode*) getLeftSubtree())= *((LFullAVLBTreeNode*)rNode.getLeftSubtree());
  }

  if (!rNode.getRightSubtree()->isNull())
  {
    linkRightSubtree(new LFullAVLBTreeNode(rNode.getRightSubtree()->getSelfData()));
    *((LFullAVLBTreeNode*) getRightSubtree())= *((LFullAVLBTreeNode*)rNode.getRightSubtree());
  }

  flag= 0;

  return *this;
}


// ---------------------------------------------------------------
// Function:       singleRotateLeft()
// Descritpion:    Do a single left rotation of the tree
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
LFullAVLBTreeNode* LFullAVLBTreeNode::singleRotateLeft
  (void)
{
  LFullAVLBTreeNode* pNewRoot= (LFullAVLBTreeNode *) getLeftSubtree();

  linkLeftSubtree(pNewRoot->getRightSubtree());
  pNewRoot->linkRightSubtree(this);
  calculateHeight();
  pNewRoot->calculateHeight();

  return pNewRoot;
}


// ---------------------------------------------------------------
// Function:       singleRotateRight()
// Descritpion:    Do a single right rotation of the tree
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
LFullAVLBTreeNode* LFullAVLBTreeNode::singleRotateRight
  (void)
{
  LFullAVLBTreeNode* pNewRoot= (LFullAVLBTreeNode *) getRightSubtree();
```

```
  linkRightSubtree(pNewRoot->getLeftSubtree());
  pNewRoot->linkLeftSubtree(this);
  calculateHeight();
  pNewRoot->calculateHeight();
  return pNewRoot;

}


// ---------------------------------------------------------
// Function:       doubleRotateLeft()
// Descritpion:    Do the left-right double rotation
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
LFullAVLBTreeNode* LFullAVLBTreeNode::doubleRotateLeft
  (void)
{
  linkLeftSubtree(((LFullAVLBTreeNode *) getLeftSubtree())->singleRotateRight());
  return singleRotateLeft();
}


// ---------------------------------------------------------
// Function:       doubleRotateRight()
// Descritpion:    Do the right-left double rotation
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
LFullAVLBTreeNode* LFullAVLBTreeNode::doubleRotateRight
  (void)
{
  linkRightSubtree(((LFullAVLBTreeNode *) getRightSubtree())->singleRotateLeft());
  return singleRotateRight();
}


// ---------------------------------------------------------
// Function:       detachSmallestNode()
// Descritpion:    Detach the smallest (deepest left child) node
//                 from the binary tree and re-connect the
//                 rest of the binary tree. The detached node is
//                 returned.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
LFullAVLBTreeNode* LFullAVLBTreeNode::detachSmallestNode
  (void)
{
  LTmplStack<LFullAVLBTreeNode*> stackTrace;

  if (getLeftSubtree()->isNull())
  {
    // ------------------------------------------------
    // Since all the data items in the right subtree
    // is greater than m_data, the smallest node is
    // this node.
    // ------------------------------------------------
    return this;
  }

  // ------------------------------------------------
  // Find the leftmost node which is the smallest node.
  // ------------------------------------------------

  LFullAVLBTreeNode* pSmallest= (LFullAVLBTreeNode*) getLeftSubtree();
  LFullAVLBTreeNode* pSmallestParent= this;

  stackTrace.push(pSmallestParent);
```

```
  while (!pSmallest->getLeftSubtree()->isNull())
  {
    pSmallestParent= pSmallest;
    stackTrace.push(pSmallestParent);
    pSmallest= (LFullAVLBTreeNode*) pSmallest->getLeftSubtree();
  }

  // ------------------------------------------------
  // Detach the smallest node from the tree. Reconnect
  // the rest of the tree. Return the detached node.
  // ------------------------------------------------

  pSmallestParent->linkLeftSubtree(pSmallest->getRightSubtree());
  pSmallest->linkRightSubtree(this);

  // ------------------------------------------------------
  // If the tree is un-balanced after detaching the smallest
  // node. then re-balance the tree.
  // ------------------------------------------------------

  LFullAVLBTreeNode* pNode= stackTrace.pop();
  LFullAVLBTreeNode* pParentNode;

  while (!stackTrace.isEmpty())
  {
    pParentNode= stackTrace.pop();
    pParentNode->linkLeftSubtree(pNode->balance('R'));
    pNode= pParentNode;
    pParentNode->calculateHeight();
  }

  // ---------------------------------------
  // pSmallest is now the parent node of pNode.
  // ---------------------------------------

  pSmallest->linkRightSubtree(pNode->balance('R'));

  return pSmallest;
}



// ------------------------------------------------------------
// Function:       calculateHeight()
// Descritpion:    Re-calculate the height of the tree


// Errors Generated: None
// Limitations:    none.
// ------------------------------------------------------------
void LFullAVLBTreeNode::calculateHeight
  (void)
{
  const int lh= getLeftSubtree()->getHeight();
  const int rh= getRightSubtree()->getHeight();
  m_height= (MAX(lh, rh) + 1);
}


// ------------------------------------------------------------
// Function:       banlance
// Descritpion:    Balnace the tree. Return the new root.


// Errors Generated: None
// Limitations:    none.
// ------------------------------------------------------------
LFullAVLBTreeNode* LFullAVLBTreeNode::balance
  (const char direction)
{
```

```
  LFullAVLBTreeNode* pNewRoot= this;

  if (direction == 'L')
  {
    if ((getLeftSubtree()->getHeight() -
       getRightSubtree()->getHeight()) == 2)
    {
      assert(!getLeftSubtree()->isNull());

      if (getLeftSubtree()->getLeftSubtree()->getHeight() >=
         getLeftSubtree()->getRightSubtree()->getHeight())
      {
        pNewRoot= singleRotateLeft();
      }
      else
      {
        pNewRoot= doubleRotateLeft();
      }
    }
    else
      calculateHeight();
  }
  else if (direction == 'R')
  {
    if ((getRightSubtree()->getHeight() -
       getLeftSubtree()->getHeight()) == 2)
    {

      assert(!getRightSubtree()->isNull());

      if (getRightSubtree()->getRightSubtree()->getHeight() >=
         getRightSubtree()->getLeftSubtree()->getHeight())
      {
        pNewRoot= singleRotateRight();
      }
      else
      {
        pNewRoot= doubleRotateRight();
      }
    }
    else
      calculateHeight();
  }
  else
  {
    assert(FALSE);
  }

 // pNewRoot->calculateHeight();

  return pNewRoot;
}



// ------------------------------------------------------------
// Function:       space()
// Description:    Return the space of the entire AVL binary tree.
//                 Note that in an application, all the empty
//                 nodes in all the binary trees are actually
//                 represented by the same static null node.
//                 Therefore we only calculate the space
//                 occupied by the null node once in the
//                 application.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
long LFullAVLBTreeNode::space
  (void) const
{
```

```
    long treeSpace= sizeof(*this);
    if (!getLeftSubtree()->isNull())
    {
      treeSpace+= getLeftSubtree()->space();
    }

    if (!getRightSubtree()->isNull())
    {
      treeSpace+= getRightSubtree()->space();
    }

    if (!LFullBTreeNode::m_bCountNullNodeSpace_S)
    {
      treeSpace+= LFullBTreeNode::getNullNode_S()->space();
      m_bCountNullNodeSpace_S= TRUE;
    }

    if (!m_bCountNullAVLNodeSpace_S)
    {
      treeSpace+= m_nullAVLNode_S.space();
      m_bCountNullAVLNodeSpace_S= TRUE;
    }
    return treeSpace;
  }



  ////////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////////
  // Implementation of class LNullAVLBTreeNode.
  //

  // ------------------------------------------------------------
  // Function:        insert
  // Descritpion:     Insert a data item into the null node.
  //                  Return a full AVL tree node.
  // Errors Generated: None
  // Limitations:     Didn't handle the memory allocation failure
  //                  if occurred.
  // ------------------------------------------------------------
  LBaseBTreeNode* LNullAVLBTreeNode::insert
    (const int data)
  {
    return new LFullAVLBTreeNode(data);
  }
```

Code Listing 15: *LTmplStack.h*

```
// ****************************************************************
// FILE:        LTmplStack.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Feb. 10, 1998
// DESCRIPTION:  Template class of a stack. Use null node method.
// ****************************************************************

#ifndef LTMPLSTACK_H
#define LTMPLSTACK_H


#include <iostream.h>
#include "LTypes.h"

// ------------------------------------------
// LTmplBaseStackNode defines the virtual APIs
// to operate on a stack node.
// ------------------------------------------
template <class DATA_TYPE>
class LTmplBaseStackNode
{

// -------------------------
// Member functions
// -------------------------
public:
  virtual DATA_TYPE   getData(void) const = 0;
  virtual BOOL        isNull(void) const = 0;
  virtual LTmplBaseStackNode<DATA_TYPE>* getNext(void) = 0;
};


// ------------------------------------------
// LTmplNullStackNode defines a null stack
// node which contains no data.
// ------------------------------------------
template <class DATA_TYPE>
class LTmplNullStackNode: public LTmplBaseStackNode <DATA_TYPE>
{
public:
  virtual DATA_TYPE   getData(void) const {cout << "ERROR: cannot get data from a null stack node." << endl; DATA_TYPE x;
return x;}
  virtual BOOL        isNull(void) const {return TRUE;}
  virtual LTmplBaseStackNode<DATA_TYPE>* getNext(void) {cout << "ERROR: cannot get next stack node from a null stack
node." << endl; return 0;}

};


// ------------------------------------------
// LTmplFullStackNode defines a full stack node
// which contains actual data. It has a static
// null stack node which is used to replace the
// functionality of a null pointer.
// ------------------------------------------
template <class DATA_TYPE>
class LTmplFullStackNode: public LTmplBaseStackNode <DATA_TYPE>
{

// -----------------------------
// Constructors and destructors
// -----------------------------
public:
  LTmplFullStackNode  (const DATA_TYPE& data, LTmplBaseStackNode<DATA_TYPE> *pNext= &m_nullNode_S) {m_data=
data; m_pNext= pNext;}
```

```
  virtual BOOL       isNull   (void) const {return FALSE;}
  virtual DATA_TYPE  getData  (void) const {return m_data;}

  virtual LTmplBaseStackNode<DATA_TYPE>*  getNext  (void) {return m_pNext;}

  virtual  LTmplNullStackNode<DATA_TYPE>*  getStaticNullNode (void)  {return &m_nullNode_S;}
  static   LTmplNullStackNode<DATA_TYPE>*  getNullNode_S (void) {return &m_nullNode_S;}

private:
  DATA_TYPE m_data;  // data item of the node.
  LTmplBaseStackNode<DATA_TYPE>* m_pNext;

  static LTmplNullStackNode<DATA_TYPE> m_nullNode_S;

};


// ------------------------------------------
// LTmplStack defines a simple stack.
// ------------------------------------------
template <class DATA_TYPE>
class LTmplStack
{
public:
 LTmplStack  (void);
  void          push      (const DATA_TYPE data);
  DATA_TYPE     pop       (void);
  int           getCount  (void) const {return m_count;}
  void          traverse  (void) const;
  BOOL          isEmpty   (void) const { return (m_count == 0);}

private:
  LTmplBaseStackNode<DATA_TYPE>* m_pTop;
  int m_count;

};

////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
// Implementation

// ------------------------------------------------------------
// Function:      LTmplStack()
// Descritpion:    Default constructor.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
template <class DATA_TYPE>
LTmplStack<DATA_TYPE>::LTmplStack(void)
{
  m_pTop= LTmplFullStackNode<DATA_TYPE>::getNullNode_S();
  m_count= 0;
}


// ------------------------------------------------------------
// Function:      push()
// Descritpion:    Push a data item into the stack.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
template <class DATA_TYPE>
void LTmplStack<DATA_TYPE>::push
  (const DATA_TYPE data)
{
  LTmplFullStackNode<DATA_TYPE> *pNewNode= new LTmplFullStackNode<DATA_TYPE>(data, m_pTop);
  m_pTop= pNewNode;
  m_count++;
}
```

```
// ---------------------------------------------------------
// Function:       pop()
// Description:    Pop a data item from the stack.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
DATA_TYPE LTmplStack<DATA_TYPE>::pop(void)
{
  DATA_TYPE data= m_pTop->getData();
  LTmplBaseStackNode<DATA_TYPE>* pOld= m_pTop;
  m_pTop= m_pTop->getNext();
  delete pOld;
  m_count--;
  return data;
}



// ---------------------------------------------------------
// Function:       traverse()
// Description:    Display all the data in the stack.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
void LTmplStack<DATA_TYPE>::traverse(void) const
{
  LTmplBaseStackNode<DATA_TYPE>* pNode= m_pTop;

  int i= 0;

  cout << "traverse stack." << endl;
  while (!pNode->isNull())
  {
    i++;
    cout << pNode->getData() << "\t";
    if (i % 4 == 0)
      cout << endl;

    pNode= pNode->getNext();
  }
  cout << endl << endl;
}

#endif // LTMPLSTACK_H
```

Code Listing 16: *LPtrAVLBTreeNode.h*

```
// ****************************************************************
// FILE:        LPtrAVLBTreeNode.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Feb. 10, 1998
// DESCRIPTION:  This file defines class LPtrAVLBTreeNode
// ****************************************************************

#ifndef LPTRAVLBTREENODE_H
#define LPTRAVLBTREENODE_H

#include "LPtrBTreeNode.h"


// ------------------------------------------------
// Class LPtrAVLBTreeNode implements a node for an
// AVL tree.
// ------------------------------------------------
class LPtrAVLBTreeNode: public LPtrBTreeNode
{

// -----------------------------------------
// Constructors, destructors and enum types.
// -----------------------------------------
public:
  LPtrAVLBTreeNode(const int newData);
  LPtrAVLBTreeNode(const LPtrAVLBTreeNode& rNode) : LPtrBTreeNode(rNode) {*this= rNode;}

// ---------
// Methods
// ---------
public:
  virtual LPtrBTreeNode* insert(const int data);
  LPtrBTreeNode* remove(const int data, BOOL& bDeleteThis, BOOL& bRemoveOK);

// operators
  LPtrAVLBTreeNode& operator= (const LPtrAVLBTreeNode& rNode);

  // inlines
  virtual int getHeight(void) const {return m_height;}
  void      setHeight(const int height) { m_height= height;}
  virtual long space (void) const;

public:
  void          calculateHeight   (void);
  LPtrAVLBTreeNode*  balance          (const char direction);

private:
  LPtrAVLBTreeNode* detachSmallestNode  (void);
  LPtrAVLBTreeNode* singleRotateLeft    (void);
  LPtrAVLBTreeNode* doubleRotateLeft    (void);
  LPtrAVLBTreeNode* singleRotateRight   (void);
  LPtrAVLBTreeNode* doubleRotateRight   (void);

// ------------
// Attributes
// ------------
private:
  int               m_height;
};

#endif //LPTRAVLBTREENODE_H
```

Code Listing 17: *LPtrAVLBTreeNode.cpp*

```
// ****************************************************************
// FILE:        LPtrAVLBTreeNode.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Feb. 10. 1998
// DESCRIPTION:  Refer to LPtrAVLBTreeNode.h
// ****************************************************************

#define LPTRAVLBTREENODE_CPP
// -------------
// Includes
// -------------

#include <iostream.h>
#include <assert.h>

#include "LPtrAVLBTreeNode.h"
#include "LTmplPtrStack.h"

#define MAX(a, b) ( ((a) >= (b))? (a) : (b) )

// ------------------------------------------------
// Initialization of the static member variables.
// ------------------------------------------------

// ------------------------------------------------------------
// Function:       LPtrAVLBTreeNode()
// Descritpion:    Constructor.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
LPtrAVLBTreeNode::LPtrAVLBTreeNode
  (const int newData)
  :LPtrBTreeNode(newData),
   m_height(0)
{
  return;
}

// ------------------------------------------------------------
// Function:       insert()
// Descritpion:    Insert a data item into the binary tree.
//                 Return the new root for the binary tree.
//                 Re-balance the tree when needed.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
LPtrBTreeNode* LPtrAVLBTreeNode::insert
  (const int data) // new data item to be inserted
{
  LPtrAVLBTreeNode* pNewRoot= this;

  if ( data < getSelfData() )
  {
    if (getLeftSubtree() != NULL)
    {
      linkLeftSubtree(getLeftSubtree()->insert(data));
    }
    else
    {
      linkLeftSubtree(new LPtrAVLBTreeNode(data));
    }

    pNewRoot= balance('L');
  }
  else if ( data > getSelfData() )
  {
```

```
    if (getRightSubtree() != NULL)
    {
      linkRightSubtree(getRightSubtree()->insert(data));
    }
    else
    {
      linkRightSubtree(new LPtrAVLBTreeNode(data));
    }

    pNewRoot= balance('R');
  }
  // else m_data == getSelfData(). Don't do anything.

  return pNewRoot;
}

// ------------------------------------------------------------
// Function:       remove()
// Description:    Remove a data item from the binary tree.
//                 Return the new root for the binary tree.
//                 Re-balance the tree when needed.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
LPtrBTreeNode* LPtrAVLBTreeNode::remove
  (const int  data,        // data to be removed
   BOOL&      bDeleteThis,  // TRUE if this node itself need to be deleted.
   BOOL&      bRemoveOK)    // TRUE if we sucessfully remove the data item from the tree
{

  LPtrBTreeNode* pNewRoot;

  bDeleteThis= FALSE;
  bRemoveOK= FALSE;

  if ( data == getSelfData())
  {
    // ---------------------------------------------
    // This node is the node which contains the
    // data item to be removed. Set bDeleteThis to
    // TRUE and bRemoveOK to TRUE.
    // ---------------------------------------------
    bDeleteThis= TRUE;
    bRemoveOK= TRUE;

    if (getLeftSubtree() == NULL)
    {
      pNewRoot= getRightSubtree(); // new root is the right subtree.
    }
    else if (getRightSubtree() == NULL)
    {
      pNewRoot= getLeftSubtree(); // new root is the left subtree.
    }
    else
    {
      // ----------------------------------
      // None of the left and right subtree
      // is Null, we should replace the root
      // with the smallest node of the right
      // subtree.
      // ----------------------------------

      LPtrAVLBTreeNode* pNewThisNode;

      pNewThisNode= ((LPtrAVLBTreeNode*) getRightSubtree())->detachSmallestNode();

      // Connect the new root with the left subtree.
      pNewThisNode->linkLeftSubtree(getLeftSubtree());
      pNewThisNode->calculateHeight();
```

```
      // ---------------------------------
      // Re-balance the new tree.
      // ---------------------------------

      pNewThisNode= pNewThisNode->balance('L');
      pNewThisNode->calculateHeight();
      pNewRoot= pNewThisNode;
    }
  }
  else if ( data < getSelfData())
  {
    if (getLeftSubtree() == NULL)
    {
      bRemoveOK= FALSE;
    }
    else
    {
      BOOL bDeleteLeft;
      LPtrBTreeNode* pOldLeft= getLeftSubtree();

      linkLeftSubtree(getLeftSubtree()->remove(data, bDeleteLeft, bRemoveOK));

      if (bDeleteLeft)
      {
        delete pOldLeft;
      }
    }

    // ---------------------------------------------------
    // Data item is removed from the left subtree. The tree
    // root is still this node.
    // ---------------------------------------------------

    pNewRoot= this;

    // ---------------------------------------------------
    // If the data is successfully deleted, and the tree is
    // not balanced yet, we should re-balance the tree.
    // Note that deleting a node from the left sub-tree is
    // equivalent to inserting a node in the right sub-tree.
    // ---------------------------------------------------

    if (bRemoveOK )
    {
      pNewRoot= balance('R');
    }
  }
  else if (data > getSelfData())
  {
    if (getRightSubtree() == NULL)
    {
      bRemoveOK= FALSE;
    }
    else
    {
      BOOL bDeleteRight;
      LPtrBTreeNode* pOldRight= getRightSubtree();

      linkRightSubtree(getRightSubtree()->remove(data, bDeleteRight, bRemoveOK));
      if (bDeleteRight)
      {
        delete pOldRight;
      }
    }

    // ---------------------------------------------------
    // Data item is removed from the right subtree. The tree
    // root is still this node.
    // ---------------------------------------------------
```

```
    pNewRoot= this;

    // ------------------------------------------------------
    // If the data is successfully deleted, and the tree is
    // not balanced yet, we should re-balance the tree.
    // Note that deleting a node from the right sub-tree is
    // equivalent to inserting a node in the left sub-tree.
    // ------------------------------------------------------

    if (bRemoveOK)
    {
      pNewRoot= balance('L');
    }
  }
  return pNewRoot;
}



// ------------------------------------------------------------------
// Function:       operator= ()
// Descritpion:    Copy the entire binary tree, including the
//                 subtrees. New memory is allocated.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------------
LPtrAVLBTreeNode& LPtrAVLBTreeNode::operator=
  (const LPtrAVLBTreeNode& rNode)
{

  static int flag= 0;
  // ----------------------------
  // Delete the existing tree.
  // ----------------------------

  if (flag == 0)
  {
    removeSubtrees();
    flag= 1;
    setSelfData(rNode.getSelfData());
    m_height= rNode.getHeight();
  }

  if (rNode.getLeftSubtree() != NULL)
  {
    linkLeftSubtree(new LPtrAVLBTreeNode(rNode.getLeftSubtree()->getSelfData()));
    *((LPtrAVLBTreeNode*) getLeftSubtree())= *((LPtrAVLBTreeNode*)rNode.getLeftSubtree());
  }

  if (rNode.getRightSubtree() != NULL)
  {
    linkRightSubtree(new LPtrAVLBTreeNode(rNode.getRightSubtree()->getSelfData()));
    *((LPtrAVLBTreeNode*) getRightSubtree())= *((LPtrAVLBTreeNode*)rNode.getRightSubtree());
  }

  flag= 0;

  return *this;
}


// ------------------------------------------------------------------
// Function:       singleRotateLeft()
// Descritpion:    Do a single left rotation of the tree
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------------
LPtrAVLBTreeNode* LPtrAVLBTreeNode::singleRotateLeft
  (void)
{
```

```
    LPtrAVLBTreeNode* pNewRoot= (LPtrAVLBTreeNode *) getLeftSubtree();

    linkLeftSubtree(pNewRoot->getRightSubtree());
    pNewRoot->linkRightSubtree(this);
    calculateHeight();
    pNewRoot->calculateHeight();

    return pNewRoot;
  }


  // ------------------------------------------------------------
  // Function:       singleRotateRight()
  // Description:    Do a single right rotation of the tree
  // Errors Generated: None
  // Limitations:
  // ------------------------------------------------------------
  LPtrAVLBTreeNode* LPtrAVLBTreeNode::singleRotateRight
    (void)
  {
    LPtrAVLBTreeNode* pNewRoot= (LPtrAVLBTreeNode *) getRightSubtree();

    linkRightSubtree(pNewRoot->getLeftSubtree());
    pNewRoot->linkLeftSubtree(this);
    calculateHeight();
    pNewRoot->calculateHeight();
    return pNewRoot;

  }


  // ------------------------------------------------------------
  // Function:       doubleRotateLeft()
  // Description:    Do the left-right double rotation
  // Errors Generated: None
  // Limitations:
  // ------------------------------------------------------------
  LPtrAVLBTreeNode* LPtrAVLBTreeNode::doubleRotateLeft
    (void)
  {
    linkLeftSubtree(((LPtrAVLBTreeNode *) getLeftSubtree())->singleRotateRight());
    return singleRotateLeft();
  }


  // ------------------------------------------------------------
  // Function:       doubleRotateRight()
  // Description:    Do the right-left double rotation
  // Errors Generated: None
  // Limitations:
  // ------------------------------------------------------------
  LPtrAVLBTreeNode* LPtrAVLBTreeNode::doubleRotateRight
    (void)
  {
    linkRightSubtree(((LPtrAVLBTreeNode *) getRightSubtree())->singleRotateLeft());
    return singleRotateRight();
  }


  // ------------------------------------------------------------
  // Function:       detachSmallestNode()
  // Description:    Detach the smallest (deepest left child) node
  //                 from the binary tree and re-connect the
  //                 rest of the binary tree. The detached node is
  //                 returned.
  // Errors Generated: None
  // Limitations:
  // ------------------------------------------------------------
  LPtrAVLBTreeNode* LPtrAVLBTreeNode::detachSmallestNode
    (void)
```

```
{
  LTmplPtrStack<LPtrAVLBTreeNode*> stackTrace;

  if (getLeftSubtree() == NULL)
  {
    // ------------------------------------------------
    // Since all the data items in the right subtree
    // is greater than m_data, the smallest node is
    // this node.
    // ------------------------------------------------
    return this;
  }

  // ------------------------------------------------
  // Find the leftmost node which is the smallest node.
  // ------------------------------------------------

  LPtrAVLBTreeNode* pSmallest= (LPtrAVLBTreeNode*) getLeftSubtree();
  LPtrAVLBTreeNode* pSmallestParent= this;

  stackTrace.push(pSmallestParent);
  while (pSmallest->getLeftSubtree() != NULL)
  {
    pSmallestParent= pSmallest;
    stackTrace.push(pSmallestParent);
    pSmallest= (LPtrAVLBTreeNode*) pSmallest->getLeftSubtree();
  }

  // ------------------------------------------------
  // Detach the smallest node from the tree. Reconnect
  // the rest of the tree. Return the detached node.
  // ------------------------------------------------

  pSmallestParent->linkLeftSubtree(pSmallest->getRightSubtree());
  pSmallest->linkRightSubtree(this);

  // ------------------------------------------------------
  // If the tree is un-balanced after detaching the smallest
  // node, then re-balance the tree.
  // ------------------------------------------------------

  LPtrAVLBTreeNode* pNode= stackTrace.pop();
  LPtrAVLBTreeNode* pParentNode;

  while (!stackTrace.isEmpty())
  {
    pParentNode= stackTrace.pop();
    pParentNode->linkLeftSubtree(pNode->balance('R'));
    pNode= pParentNode;
    pParentNode->calculateHeight();
  }

  // ------------------------------------------
  // pSmallest is now the parent node of pNode.
  // ------------------------------------------

  pSmallest->linkRightSubtree(pNode->balance('R'));

  return pSmallest;
}


// ------------------------------------------------------------
// Function:       calculateHeight()
// Description:    Re-calculate the height of the tree


// Errors Generated: None
// Limitations:    none.
```

```
// ----------------------------------------------------------
void LPtrAVLBTreeNode::calculateHeight
  (void)
{
  const int lh= (getLeftSubtree() == NULL) ? -1 : ((LPtrAVLBTreeNode*) getLeftSubtree())->getHeight();
  const int rh= (getRightSubtree() == NULL) ? -1 : ((LPtrAVLBTreeNode*) getRightSubtree())->getHeight();
  m_height= (MAX(lh, rh) + 1);
}


// ----------------------------------------------------------
// Function:       banlance
// Descritpion:    Balnace the tree. Return the new root.


// Errors Generated: None
// Limitations:    none.
// ----------------------------------------------------------
LPtrAVLBTreeNode* LPtrAVLBTreeNode::balance
  (const char direction)
{
  LPtrAVLBTreeNode* pNewRoot= this;

  const int leftHeight= (getLeftSubtree() == NULL) ? -1 : ((LPtrAVLBTreeNode*) getLeftSubtree())->getHeight();
  const int rightHeight= (getRightSubtree() == NULL) ? -1 : ((LPtrAVLBTreeNode*) getRightSubtree())->getHeight();

  if (direction == 'L')
  {
    if ((leftHeight - rightHeight) == 2)
    {
      assert(getLeftSubtree() != NULL);

      const int leftLeftHeight= (getLeftSubtree()->getLeftSubtree() == NULL) ? -1 : ((LPtrAVLBTreeNode*) getLeftSubtree()-
>getLeftSubtree())->getHeight();
      const int leftRightHeight= (getLeftSubtree()->getRightSubtree() == NULL) ? -1 : ((LPtrAVLBTreeNode*) getLeftSubtree()-
>getRightSubtree())->getHeight();

      if (leftLeftHeight >= leftRightHeight)
      {
        pNewRoot= singleRotateLeft();
      }
      else
      {
        pNewRoot= doubleRotateLeft();
      }
    }
    else
      calculateHeight();
  }
  else if (direction == 'R')
  {
    if ((rightHeight - leftHeight) == 2)
    {
      assert(getRightSubtree() != NULL);

      const int rightRightHeight= (getRightSubtree()->getRightSubtree() == NULL) ? -1 : ((LPtrAVLBTreeNode*) getRightSubtree()-
>getRightSubtree())->getHeight();
      const int rightLeftHeight= (getRightSubtree()->getLeftSubtree() == NULL) ? -1 : ((LPtrAVLBTreeNode*) getRightSubtree()-
>getLeftSubtree())->getHeight();

      if (rightRightHeight >= rightLeftHeight)
      {
        pNewRoot= singleRotateRight();
      }
      else
      {
        pNewRoot= doubleRotateRight();
      }
    }
    else
```

```
      calculateHeight();
    }
    else
    {
      assert(FALSE);
    }

    return pNewRoot;
}


// ------------------------------------------------------------
// Function:        space()
// Descritpion:     Return the space needed for the entire AVL tree.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
long LPtrAVLBTreeNode::space
  (void) const
{

  long treeSpace= sizeof(*this);

  if (getLeftSubtree() != NULL)
  {
    treeSpace+= getLeftSubtree()->space();
  }

  if (getRightSubtree() != NULL)
  {
    treeSpace+= getRightSubtree()->space();
  }

  return treeSpace;
}
```

Code Listing 18: *LTmplPtrStack.h*

```
// ****************************************************************
// FILE:        LTmplPtrStack.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Feb. 15, 1998
// DESCRIPTION:  Template class of a stack. Use null pointer
// ****************************************************************

#ifndef LTMPLPTRSTACK_H
#define LTMPLPTRSTACK_H


#include <iostream.h>
#include "LTypes.h"


// ------------------------------------------------
// LTmplPtrStackNode defines a stack node. It
// uses a null pointer.
// ------------------------------------------------
template <class DATA_TYPE>
class LTmplPtrStackNode
{

// ------------------------------
// Constructors and destructors
// ------------------------------
public:
  LTmplPtrStackNode  (const DATA_TYPE& data, LTmplPtrStackNode<DATA_TYPE> *pNext= NULL) {m_data= data; m_pNext=
pNext;}
  LTmplPtrStackNode  (void) {m_pNext= NULL;}

  virtual BOOL          isNull  (void) const {return FALSE;}
  virtual DATA_TYPE     getData (void) const {return m_data;}

  virtual LTmplPtrStackNode<DATA_TYPE>*   getNext (void) {return m_pNext;}

private:
  DATA_TYPE m_data;  // data item of the node.
  LTmplPtrStackNode<DATA_TYPE>* m_pNext;

};


// ------------------------------------------
// LTmplPtrStack defines a simple stack.
// ------------------------------------------
template <class DATA_TYPE>
class LTmplPtrStack
{
public:
   LTmplPtrStack (void);
  void          push     (const DATA_TYPE data);
  DATA_TYPE     pop      (void);
  int           getCount  (void) const {return m_count;}
  void          traverse  (void) const;
  BOOL          isEmpty   (void) const { return (m_count == 0);}

private:
  LTmplPtrStackNode<DATA_TYPE>* m_pTop;
  int m_count;

};

////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
```

```
// Implementation

// ---------------------------------------------------------
// Function:       LTmplPtrStack()
// Descritpion:    Default constructor.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
LTmplPtrStack<DATA_TYPE>::LTmplPtrStack(void)
{
  m_pTop= NULL;
  m_count= 0;
}


// ---------------------------------------------------------
// Function:       push()
// Descritpion:    Push a data item into the stack.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
void LTmplPtrStack<DATA_TYPE>::push
  (const DATA_TYPE data)
{
  LTmplPtrStackNode<DATA_TYPE> *pNewNode= new LTmplPtrStackNode<DATA_TYPE>(data, m_pTop);
  m_pTop= pNewNode;
  m_count++;
}


// ---------------------------------------------------------
// Function:       pop()
// Descritpion:    Pop a data item from the stack.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
DATA_TYPE LTmplPtrStack<DATA_TYPE>::pop(void)
{
  DATA_TYPE data= m_pTop->getData();
  LTmplPtrStackNode<DATA_TYPE>* pOld= m_pTop;
  m_pTop= m_pTop->getNext();
  delete pOld;
  m_count--;
  return data;
}



// ---------------------------------------------------------
// Function:       traverse()
// Descritpion:    Display all the data in the stack.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
template <class DATA_TYPE>
void LTmplPtrStack<DATA_TYPE>::traverse(void) const
{
  LTmplPtrStackNode<DATA_TYPE>* pNode= m_pTop;

  int i= 0;

  cout << "traverse stack." << endl;
  while (pNode != NULL)
  {
    i++;
    cout << pNode->getData() << "\t";
    if (i % 4 == 0)
```

```
        cout << endl;

      pNode= pNode->getNext();
    }
    cout << endl << endl;
}

#endif // LTMPLPTRSTACK_H
```

Code Listing 19: *LSListNode.h*

```
// ***************************************************************
// FILE:        LSListNode.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Sept. 02, 1997
// DESCRIPTION:  This file contains declarations of three classes:
//              LBaseSListNode, LNullSListNode and LFullSListNode.
//              Both LNullSListNode and LFullSListNode are
//              derived from LBaseSListNode. The three classes
//              are used to describe a singly linked list data
//              structure. It uses LNullSListNode to represent
//              a null node and replace the traditional NULL
//              pointer
// ***************************************************************

#ifndef LSLISTNODE_H
#define LSLISTNODE_H

#include <iostream.h>
#include <fstream.h>

//#include "LTypes.h"

// ------------------------------------------------
// Class LBaseSListNode defines a set of pure
// virual funcitons to manipulate a single
// linked list. Here we acutually view each node
// as a valid singly linked list, since through the
// node, we can access the entire linked list which
// uses this node as the head node.
// ------------------------------------------------
class LBaseSListNode
{
// --------------------------
// constructors, destructors
// and enum types.
// --------------------------

// ------------------------
// Methods
// ------------------------
public:
  // Insert a new data item in the front of the singly linked list
  virtual LBaseSListNode* insert(const int data) = 0;

  // Insert a new data item at the end of the singly linked list.
  virtual LBaseSListNode* append(const int data) = 0;

  // Remove all the data items that equal to "data" from the
  // singly linked list. Return the new head
  // for the singly linked list.
  // If this node iteself need to be removed,
  // bDeleteThis is set to TRUE, otherwise it is set to FALSE.
  // nCount return the number of data items removed.
  virtual LBaseSListNode* remove(const int data, BOOL& bDeleteThis, int& nCount) = 0;

  // Search a data item in the singly linked list. If find, return TRUE.
  // Otherwise return FALSE.
  virtual BOOL search(const int data) = 0;

  // Traverse the entire singly linked list. Prints the result to
  // to the standard output.
  virtual void traverse(void) const = 0;

  // Traverse the entire singly linked list.
  // Print the result to a file represented by the
  // ofstream.
```

```
    virtual void traverse(ofstream& ofs) const = 0;

    // Return TRUE if this node is a null node. Otherwise
    // return FALSE.
    virtual BOOL isNull(void) const = 0;

    // Return the node data.
    virtual int getSelfData(void) const = 0;

    // Link this node with another node.
    virtual void linkNext(LBaseSListNode* pNext) = 0;

    // Get the next node pointer. The next node cannot
    // be modified.
    virtual const LBaseSListNode* getNext(void) const = 0;

    // Get the next node pointer. The next node can
    // be modified.
    virtual LBaseSListNode* getNext(void) = 0;

    // Print the node data to the standard output.
    virtual void outputSelfData(void) const = 0;

    // Print the node data to a file.
    virtual void outputSelfData(ofstream& ofs) const = 0;

    // Return the space required for the entired singly linked list.
    virtual long space(void) const  = 0;
};


// ----------------------------------------------------
// Class LNullSListNode implements a null single linked
// list which has no data and the next node.
// ----------------------------------------------------
class LNullSListNode: public LBaseSListNode
{

// -------------
// Methods
// -------------
public:
    // Function insert() and append() cannot be inline function
    // they need to create a LFullSListNode.
    // But class LFullSListNode is not defined at this point.
    virtual LBaseSListNode* insert(const int data);
    virtual LBaseSListNode* append(const int data);

    // inlines
    virtual LBaseSListNode* remove(const int data, BOOL& bDeleteThis, int& nCount)
    {
      bDeleteThis= FALSE;
      nCount= 0;
      return this;
    }

    virtual BOOL   search         (const int data) {return FALSE;}
    virtual void   traverse       (void) const {}
    virtual void   traverse       (ofstream& ofs) const {}
    virtual BOOL   isNull         (void) const {return TRUE;}
    virtual int    getSelfData    (void) const { cout << "ERROR: cannot get data from a NULL singly linked list node." << endl; return
-1;}
    virtual void   linkNext       (LBaseSListNode* pNext) { cout << "ERROR: cannot set the next node for a NULL singly linked list
node." << endl;}
    virtual void   outputSelfData (void) const { cout << "NIL";}
    virtual void   outputSelfData (ofstream& ofs) const { ofs << "NIL";}
    virtual long   space(void) const {return sizeof(*this);}
    virtual const LBaseSListNode*   getNext(void) const { cout << "ERROR: cannot get the next node for a NULL singly linked list
node." << endl; return this;}
```

```
  virtual LBaseSListNode*       getNext(void) { cout << "ERROR: cannot get the next node for a NULL singly linked list node." <<
endl; return this;}
};


// ------------------------------------------------
// Class LFullSListNode implements a full single linked
// list node which has data and the next pointer.
// Since a full singly linked list node has a pointer
// to the next node in the list, it can be used to
// represent an entire linked list.
// ------------------------------------------------
class LFullSListNode: public LBaseSListNode
{

// ------------------------------------------
// Constructors, destructors and enum types.
// ------------------------------------------
public:
  LFullSListNode(const int newData, LBaseSListNode* pNext= &m_nullNode_S);
  LFullSListNode(const LFullSListNode& rNode);
  virtual ~LFullSListNode(void);

// ---------
// Methods
// ---------
public:
  virtual LBaseSListNode* insert(const int data);
  virtual LBaseSListNode* append(const int data);
  virtual LBaseSListNode* remove(const int data, BOOL& bDeleteThis, int& nCount);

  virtual BOOL   search(const int data);
  virtual void   traverse(void) const;
  virtual void   traverse(ofstream& ofs) const;
  void           removeSublist(void);
  virtual long   space(void) const;

// operators
  LFullSListNode& operator= (const LFullSListNode& rNode);

// inline methods
  virtual BOOL     isNull              (void) const { return FALSE;}
  virtual int      getSelfData         (void) const {return m_data;}
  virtual void     linkNext            (LBaseSListNode* pNext) {m_pNext= pNext;}
  virtual void     outputSelfData        (void) const { cout << m_data;}
  virtual void     outputSelfData        (ofstream& ofs) const { ofs << m_data;}

  virtual const LBaseSListNode*   getNext  (void) const { return m_pNext;}
  virtual LBaseSListNode*         getNext  (void) { return m_pNext;}

  virtual  LNullSListNode*       getStaticNullNode (void)  {return &m_nullNode_S;}
  static   LNullSListNode*       getNullNode_S (void)  {return &m_nullNode_S;}

// ------------
// Attributes
// ------------
private:
  int                 m_data;     // data kept with the node
  LBaseSListNode *m_pNext;    // next node which may point to a LNullSListNode or a LFullSListNode
  static LNullSListNode m_nullNode_S; // used to represent the end of the list. It is
                                      // is static since all LFullSListNodes in an
                                      // application can use the same null node.
  static BOOL   m_bCountNullNodeSpace_S; // TRUE if the space occupied by the null node is counted.
};

#endif //LSLISTNODE_H
```

Code Listing 20: *LSListNode.cpp*

```
// ****************************************************************
// FILE:        LSListNode.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Oct. 7, 1997
// DESCRIPTION:  Refer to LSListNode.h
// ****************************************************************

// -------------
// Includes
// -------------

#include <iostream.h>
#include <assert.h>

#include "LTypes.h"
#include "LSListNode.h"


// ------------------------------------------------
// Initialization of the static member variables.
// ------------------------------------------------

LNullSListNode LFullSListNode::m_nullNode_S;
BOOL LFullSListNode::m_bCountNullNodeSpace_S= FALSE;

// ------------------------------------------------------------
// Function:       LFullSListNode()
// Descritpion:    Constructor.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
LFullSListNode::LFullSListNode
  (const int newData,
   LBaseSListNode* pNext)
{
  m_data= newData;
  m_pNext= pNext;
}


// ------------------------------------------------------------
// Function:       LFullSListNode()
// Descritpion:    Copy constructor.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
LFullSListNode::LFullSListNode
  (const LFullSListNode& rNode)
{
  *this= rNode;
}

// ------------------------------------------------------------
// Function:       ~LFullSListNode()
// Descritpion:    Destructor.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
LFullSListNode::~LFullSListNode
  (void)
{
}


// ------------------------------------------------------------
// Function:       insert()
```

```
// Descritpion:     Insert a new node in front of the single
//                  linked list.
// Errors Generated: None
// Limitations:     None
// ----------------------------------------------------------
LBaseSListNode* LFullSListNode::insert
  (const int data)
{
  return  new LFullSListNode(data, this);
}

// ----------------------------------------------------------
// Function:        append()
// Descritpion:     Insert a new node at the end of the single
//                  linked list.
// Errors Generated: None
// Limitations:     None
// ----------------------------------------------------------
LBaseSListNode* LFullSListNode::append
  (const int data)
{

  LBaseSListNode* pNext= this;
  while (!pNext->getNext()->isNull())
  {
    pNext= pNext->getNext();
  }

  pNext->linkNext( new LFullSListNode(data));

  return this;
}


// ----------------------------------------------------------
// Function:        remove()
// Descritpion:     Remove all nodes which contain the data from
//                  the singly linked list. If "this" node
//                  is the node to be removed, then bDeleteThis
//                  is set to TRUE. nCount return the number
//                  of the node removed.
// Errors Generated: None
// Limitations:     None
// ----------------------------------------------------------
LBaseSListNode* LFullSListNode::remove
  (const int data,
   BOOL& bDeleteThis,
   int&  nCount)
{
  bDeleteThis= FALSE;
  nCount= 0;

  if (data == m_data)
  {
    nCount++;
    bDeleteThis= TRUE;
  }

  LBaseSListNode *pNode,
                 *pNextNode;

  while ((!m_pNext->isNull()) &&
         (data == m_pNext->getSelfData()))
  {
    pNextNode= m_pNext->getNext();
    delete m_pNext;
    nCount++;
    m_pNext= pNextNode;
  }
```

```
    if (!m_pNext->isNull())
    {
      pNode= m_pNext;
      pNextNode= pNode->getNext();

      while (!pNextNode->isNull())
      {
        if (data == pNextNode->getSelfData())
        {
          pNode->linkNext(pNextNode->getNext());
          delete pNextNode;
          nCount++;
          pNextNode= pNode->getNext();
        }
        else
        {
          pNode= pNextNode;
          pNextNode= pNextNode->getNext();
        }
      }
    }

    if (bDeleteThis)
    {
      return m_pNext;
    }
    else
    {
      return this;
    }
}


// ------------------------------------------------------------
// Function:       search()
// Descritpion:    Search the data in the singly linked list.
//                 If find the data, return TRUE, otherwise
//                 return FALSE.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
BOOL LFullSListNode::search
  (const int data)
{
  if (m_data == data)
    return TRUE;

  LBaseSListNode *pNext= m_pNext;

  while (!pNext->isNull())
  {
    if (data == pNext->getSelfData())
      return TRUE;

    pNext= pNext->getNext();
}
  return FALSE;
}

// ------------------------------------------------------------
// Function:       traverse()
// Descritpion:    Print data in the singly linked list
//                 to the standard output.
// Errors Generated: None
// Limitations:    None
// ------------------------------------------------------------
void LFullSListNode::traverse
  (void) const
{
```

```
    int i= 1;
    outputSelfData();
    cout << "\t\t";

    LBaseSListNode *pNext= m_pNext;

    while (!pNext->isNull())
    {
      pNext->outputSelfData();
      cout << "\t\t";
      i++;

      if ((i % 4) == 0)
      {
        cout << endl;
      }
      pNext= pNext->getNext();
    }
  }

  // ---------------------------------------------------------
  // Function:       traverse()
  // Descritpion:    Print data in the singly linked list
  //                 to a file stream.
  // Errors Generated: None
  // Limitations:    None
  // ---------------------------------------------------------
  void LFullSListNode::traverse
    (ofstream& ofs) const
  {
    int i= 1;
    outputSelfData(ofs);
    ofs << "\t\t";

    LBaseSListNode *pNext= m_pNext;

    while (!pNext->isNull())
    {

      pNext->outputSelfData(ofs);
      ofs << "\t\t";
      i++;

      if ((i % 4) == 0)
      {
        ofs << endl;
      }

      pNext= pNext->getNext();
    }

  }


  // ---------------------------------------------------------
  // Function:       operator =()
  // Descritpion:    copy the entire list, including next nodes.
  //                 New memory is allocated.
  // Errors Generated: None
  // Limitations:    None
  // ---------------------------------------------------------
  LFullSListNode& LFullSListNode::operator=
    (const LFullSListNode& rNode)
  {
    removeSublist();
    m_data= rNode.getSelfData();

    const LBaseSListNode  *pNodeNext= rNode.getNext();
    LBaseSListNode        *pNext= this;
```

```
    while (!pNodeNext->isNull())
    {
      pNext->linkNext(new LFullSListNode(pNodeNext->getSelfData()));
      pNext= pNext->getNext();
      pNodeNext= pNodeNext->getNext();
    }

    return *this;

  }


  // ------------------------------------------------------------
  // Function:       removeSublist()
  // Descritpion:    Remove the sublists from the single linked
  //                 list. Free the memory allocated.
  // Errors Generated: None
  // Limitations:    None
  // ------------------------------------------------------------
  void LFullSListNode::removeSublist
    (void)
  {
    LBaseSListNode  *pNext= m_pNext,
                    *pPrevNext;
    while (!pNext->isNull())
    {
      pPrevNext= pNext;
      pNext= pNext->getNext();
      delete pPrevNext;
    }

    m_pNext= getStaticNullNode();
  }


  // ------------------------------------------------------------
  // Function:       space()
  // Descritpion:    Calculate the entire space occupied by
  //                 by the singly linked list.
  // Errors Generated: None
  // Limitations:    None
  // ------------------------------------------------------------
  long LFullSListNode::space
    (void) const
  {
    long spaceNeeded= sizeof(*this);
    LBaseSListNode *pNext= m_pNext;

    while (!pNext->isNull())
    {
      spaceNeeded+= sizeof(* ((LFullSListNode *) pNext));
      pNext= pNext->getNext();
    }

    if (!m_bCountNullNodeSpace_S)
    {
      spaceNeeded+= sizeof(*pNext);
      m_bCountNullNodeSpace_S= TRUE;
    }

    return spaceNeeded;
  }



  ////////////////////////////////////////////////////////////
  ////////////////////////////////////////////////////////////
  // Implementation for class LNullSListNode
  //
```

```
// ---------------------------------------------------------
// Function:        insert()
// Description:     Insert a node into the null list. Return
//                  pointer to a LFullSListNode
// Errors Generated: None
// Limitations:     None
// ---------------------------------------------------------
LBaseSListNode* LNullSListNode::insert
  (const int data)
{
  return new LFullSListNode(data);
}


// ---------------------------------------------------------
// Function:        append()
// Description:     Append a node into the null list. Return
//                  pointer to a LFullSListNode
// Errors Generated: None
// Limitations:     None
// ---------------------------------------------------------
LBaseSListNode* LNullSListNode::append
  (const int data)
{
  return new LFullSListNode(data);
}
```

Code Listing 21: *LSList.h*

```
// ****************************************************************
// FILE:        LSList.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Nov. 01, 1997
// DESCRIPTION:  LSList encapsulates a singly linked list data
//               structure. It uses a LBaseSListNode object as
//               its root.
// ****************************************************************

#ifndef LSLIST_H
#define LSLIST_H

#include <fstream.h>

#include "LTypes.h"
#include "LSListNode.h"

class LSList
{
// ----------------------------
// Constructors, destructors and
// enum types
// ----------------------------

public:
  LSList(void);
  LSList(const LSList& rList);
  virtual ~LSList(void);

// -------------------------
// Member functions
// -------------------------
public:
  int remove (const int data);
  void empty(void);

// operators
  LSList& operator= (const LSList& rList);

// inline functions
  void insert (const int data) { m_pHead= m_pHead->insert(data);}
  void append (const int data) { m_pHead= m_pHead->append(data);}
  void traverse(void) const { m_pHead->traverse();}
  void traverse(ofstream& ofs) const { m_pHead->traverse(ofs);}
  const LBaseSListNode& getHead(void) const {return *m_pHead;}
  BOOL search (const int data) const { return m_pHead->search(data);}
  long space(void) const { return (sizeof(*this) + m_pHead->space());}

// ----------------------
// Attributes
// ----------------------
private:
  LBaseSListNode* m_pHead;
};

#endif // LSLIST_H
```

Code Listing 22: *LSList.cpp*

```
// ****************************************************************
// FILE:        LSList.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Sept. 15, 1997
// DESCRIPTION:  Refer to LSList.h
// ****************************************************************

#include <assert.h>
#include "LSList.h"
#include "LSListNode.h"

// ----------------------------------------------------------
// Function:        LSList()
// Descritpion:     Default constructor. Initialize the head
//                  to be the static LNulLSListNode member
//                  variable defined in LFullSListNode class.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LSList::LSList
  (void)
{
  m_pHead= LFullSListNode::getNullNode_S();
}

// ----------------------------------------------------------
// Function:        LSList()
// Descritpion:     Copy constructor.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LSList::LSList
  (const LSList& rList)
{
  *this= rList;
}

// ----------------------------------------------------------
// Function:        ~LSList()
// Descritpion:     Destructor. Delete the entire binary list.
//                  Free memory allocated.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LSList::~LSList(void)
{
  empty();
}

// ----------------------------------------------------------
// Function:        remove()
// Descritpion:     Remove data from the singly linked list.
//                  Return the number of nodes being removed.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
int LSList::remove
  (const int data)
{
  BOOL bDeleteHead;
  int nCount;

  LBaseSListNode* pOldHead= m_pHead;

  m_pHead= m_pHead->remove(data, bDeleteHead, nCount);

}
```

```
    if(bDeleteHead)
    {
      delete pOldHead;
    }

    return nCount;
  }

  // ------------------------------------------------------------
  // Function:        operator= ()
  // Descritpion:     Copy the entire singly linked list. Allocate
  //                  memory as needed.
  // Errors Generated: None
  // Limitations:
  // ------------------------------------------------------------
  LSList& LSList::operator=
    (const LSList& list)
  {
    if(m_pHead->isNull())
    {
      if(!list.getHead().isNull())
      {
        m_pHead= new LFullSListNode(list.getHead().getSelfData());
        *((LFullSListNode*) m_pHead)= (LFullSListNode&)list.getHead();
      }
    }
    else
    {
      if(list.getHead().isNull())
      {
        ((LFullSListNode*) m_pHead)->removeSublist();
        delete m_pHead;

        LFullSListNode aFullNode(0);

        m_pHead= aFullNode.getStaticNullNode();
      }
      else
      {
        *((LFullSListNode*) m_pHead)= (LFullSListNode &)list.getHead();
      }
    }
    return *this;
  }

  // ------------------------------------------------------------
  // Function:        empty()
  // Descritpion:     Empty the entire singly linked list.
  //                  Set the head node be a null sinlge linked
  //                  list node.
  // Errors Generated: None
  // Limitations:
  // ------------------------------------------------------------
  void LSList::empty
    (void)
  {
    if(!m_pHead->isNull())
    {
      ((LFullSListNode*) m_pHead)->removeSublist();
      delete m_pHead;

      m_pHead= LFullSListNode::getNullNode_S();
    }
  }
```

Code Listing 23: *LPtrSListNode.h*

```
// ****************************************************************
// FILE:        LPtrSListNode.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Dec. 02, 1997
// DESCRIPTION:  This file contains declarations of class
//              LPtrSListNode which is used to describe a
//              singly linked list data structure. The
//              end of the list is represented using a
//              NULL pointer.
// ****************************************************************

#ifndef LPTRSLISTNODE_H
#define LPTRSLISTNODE_H

#include <iostream.h>
#include <fstream.h>

//#include "LTypes.h"


// ---------------------------------------------------
// Class LPtrSListNode implements a  single linked
// list node which has data and the next pointer.
// Since a full singly linked list node has a pointer
// to the next node in the list, it can be used to
// represent an entire linked list.
// ---------------------------------------------------
class LPtrSListNode
{

// ----------------------------------------
// Constructors, destructors and enum types.
// ----------------------------------------
public:
  LPtrSListNode(const int newData, LPtrSListNode *pNext= NULL);
  LPtrSListNode(const LPtrSListNode& rNode);
  virtual ~LPtrSListNode(void);


// ---------
// Methods
// ---------
public:
  LPtrSListNode*   insert   (const int data);
  LPtrSListNode*   append   (const int data);
  BOOL             search   (const int data);
  void             traverse (void) const;
  void             traverse (ofstream& ofs) const;
  void             removeSublist(void);
  long             space    (void) const;

  LPtrSListNode*   remove(const int data, BOOL& bDeleteThis, int &nCount);

// operators
  LPtrSListNode& operator= (const LPtrSListNode& rNode);

// inline methods
  BOOL  isNull       (void) const { return FALSE;}
  int   getSelfData  (void) const { return m_data;}
  void  outputSelfData (void) const { cout << m_data;}
  void  outputSelfData (ofstream& ofs) const { ofs << m_data;}

  const LPtrSListNode*   getNext  (void) const { return m_pNext;}
  LPtrSListNode*         getNext  (void) { return m_pNext;}
  void                   linkNext (LPtrSListNode* pNext) {m_pNext= pNext;}
```

```
// ------------
// Attributes
// ------------
private:
  int          m_data;      // data kept with the node
  LPtrSListNode  *m_pNext;    // next LPtrSListNode
};

#endif //LPTRSLISTNODE_H
```

Code Listing 24: *LPtrSListNode.cpp*

```
// ****************************************************************
// FILE:        LSListNode.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Dec. 02. 1997
// DESCRIPTION:  Refer to LPtrSListNode.h
// ****************************************************************

// -------------
// Includes
// -------------

#include <iostream.h>
#include <assert.h>

#include "LTypes.h"
#include "LPtrSListNode.h"

// ---------------------------------------------------------
// Function:       LPtrSListNode()
// Descritpion:    Constructor.
// Errors Generated: None
// Limitations:    None
// ---------------------------------------------------------
LPtrSListNode::LPtrSListNode
  (const int newData,
   LPtrSListNode *pNext)
{
  m_data= newData;
  m_pNext= pNext;
}


// ---------------------------------------------------------
// Function:       LPtrSListNode()
// Descritpion:    Copy constructor.
// Errors Generated: None
// Limitations:    None
// ---------------------------------------------------------
LPtrSListNode::LPtrSListNode
  (const LPtrSListNode& rNode)
{
  *this= rNode;
}


// ---------------------------------------------------------
// Function:       ~LPtrSListNode()
// Descritpion:    Destructor.
// Errors Generated: None
// Limitations:    None
// ---------------------------------------------------------
LPtrSListNode::~LPtrSListNode
  (void)
{
}

// ---------------------------------------------------------------
// Function:       append()
// Descritpion:    Append a node at the end of the single
//                 linked list.
// Errors Generated: None
// Limitations:    None
// ---------------------------------------------------------------
LPtrSListNode* LPtrSListNode::append
  (const int data)
{
```

```
  LPtrSListNode* pNext= this;
  while (pNext->getNext() != NULL)
  {
    pNext= pNext->getNext();
  }

  pNext->linkNext( new LPtrSListNode(data));

  return this ;
}


// ------------------------------------------------------------
// Function:        insert()
// Descritpion:     Insert a new node in front of the single
//                  linked list.
// Errors Generated: None
// Limitations:     None
// ------------------------------------------------------------
LPtrSListNode* LPtrSListNode::insert
  (const int data)
{
  return  new LPtrSListNode(data, this);
}

// ------------------------------------------------------------
// Function:        remove()
// Descritpion:     Remove all the nodes which contain the data
//                  from the singly linked list. If this node
//                  is the node to be removed, then bDeleteThis
//                  is set to TRUE. nCount return the number of
//                  nodes removed.
// Errors Generated: None
// Limitations:     None
// ------------------------------------------------------------
LPtrSListNode* LPtrSListNode::remove
  (const int data,
   BOOL& bDeleteThis,
   int& nCount)
{
  bDeleteThis= FALSE;
  nCount= 0;

  if (data == m_data)
  {
    nCount++;
    bDeleteThis= TRUE;
  }

  LPtrSListNode *pNode,
          *pNextNode;

  while ((m_pNext != NULL) &&
      (data == m_pNext->getSelfData()))
  {
    pNextNode= m_pNext->getNext();
    delete m_pNext;
    nCount++;
    m_pNext= pNextNode;
  }

  if (m_pNext != NULL)
  {
    pNode= m_pNext;
    pNextNode= pNode->getNext();

    while (pNextNode != NULL)
    {
      if (data == pNextNode->getSelfData())
      {
```

```
        pNode->linkNext(pNextNode->getNext());
        delete pNextNode;
        nCount++;
        pNextNode= pNode->getNext();
      }
      else
      {
        pNode= pNextNode;
        pNextNode= pNextNode->getNext();
      }
    }
  }

  if (bDeleteThis)
  {
    return m_pNext;
  }
  else
  {
    return this;
  }
}


// ------------------------------------------------------------
// Function:          search()
// Description:       Search the data in the singly linked list.
//                    If find the data, return TRUE, otherwise
//                    return FALSE.
// Errors Generated: None
// Limitations:       None
// ------------------------------------------------------------
BOOL LPtrSListNode::search
  (const int data)
{
  if (m_data == data)
    return TRUE;

  LPtrSListNode *pNext= m_pNext;

  while (pNext != NULL)
  {
    if (data == pNext->getSelfData())
      return TRUE;

    pNext= pNext->getNext();
  }
  return FALSE;
}


// ------------------------------------------------------------
// Function:          traverse()
// Description:       Print data in the singly linked list
//                    to the standard output.
// Errors Generated: None
// Limitations:       None
// ------------------------------------------------------------
void LPtrSListNode::traverse
  (void) const
{

  int i= 1;
  outputSelfData();
  cout << "\t\t";


  LPtrSListNode *pNext= m_pNext;

  while (pNext != NULL)
```

```
    {
      pNext->outputSelfData();
      cout << "\t\t";
      i++;

      if ((i % 4) == 0)
      {
        cout << endl;
      }

      pNext= pNext->getNext();
    }
  }

// ------------------------------------------------------------
// Function:        traverse()
// Descritpion:     Print data in the singly linked list
//                  to a file stream.
// Errors Generated: None
// Limitations:     None
// ------------------------------------------------------------
void LPtrSListNode::traverse
  (ofstream& ofs) const
{
  int i= 1;
  outputSelfData(ofs);
  ofs << "\t\t";

  LPtrSListNode *pNext= m_pNext;

  while (pNext != NULL)
  {

    pNext->outputSelfData(ofs);
    ofs << "\t\t";
    i++;

    if ((i % 4) == 0)
    {
      ofs << endl;
    }

    pNext= pNext->getNext();
  }
}


// ------------------------------------------------------------
// Function:        operator = ()
// Descritpion:     copy the entire list, including next nodes.
//                  New memory is allocated.
// Errors Generated: None
// Limitations:     None
// ------------------------------------------------------------
LPtrSListNode& LPtrSListNode::operator=
  (const LPtrSListNode& rNode)
{

  removeSublist();
  m_data= rNode.getSelfData();

  const LPtrSListNode  *pNodeNext= rNode.getNext();
  LPtrSListNode        *pNext= this;

  while (pNodeNext != NULL)
  {
    pNext->linkNext(new LPtrSListNode(pNodeNext->getSelfData()));
    pNext= pNext->getNext();
    pNodeNext= pNodeNext->getNext();
  }
```

```
  return *this;
}


// ------------------------------------------------------------
// Function:          removeSublist()
// Descritpion:     Remove the sublists from the single linked
//                    list. Free the memory allocated.
// Errors Generated: None
// Limitations:     None
// ------------------------------------------------------------
void LPtrSListNode::removeSublist
  (void)
{
  LPtrSListNode  *pNext= m_pNext,
          *pPrevNext;
  while (pNext != NULL)
  {
    pPrevNext= pNext;
    pNext= pNext->getNext();
    delete pPrevNext;
  }

  m_pNext= NULL;
}


// ------------------------------------------------------------
// Function:          space()
// Descritpion:     Calculate the entire space occupied by
//                    by the singly linked list.
// Errors Generated: None
// Limitations:     None
// ------------------------------------------------------------
long LPtrSListNode::space
  (void) const
{
  long spaceNeeded= sizeof(*this);
  LPtrSListNode *pNext= m_pNext;

  while (pNext != NULL)
  {
    spaceNeeded+= sizeof(*pNext);
    pNext= pNext->getNext();

  }

  return spaceNeeded;
}
```

Code Listing 25: *LPtrSList.h*

```
// ****************************************************************
// FILE:            LPtrSList.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Dec. 02. 1997
// DESCRIPTION:  LPtrSList encapsulates a singly linked list data
//                    structure. It uses a LPtrSListNode object as
//                    its root.
// ****************************************************************

#ifndef LPTRSLIST_H
#define LPTRSLIST_H

#include <fstream.h>

#include "LTypes.h"
#include "LPtrSListNode.h"

class LPtrSList
{
// ----------------------------
// Constructors, destructors and
// enum types
// ----------------------------

public:
  LPtrSList(void);
  LPtrSList(const LPtrSList& rList);
  virtual ~LPtrSList(void);

// -------------------------
// Member functions
// -------------------------
public:
  int remove (const int data);
  void empty(void);

// operators
  LPtrSList& operator= (const LPtrSList& rList);

// inline functi  ons
  void insert (const int data);
  void append (const int data);
  void traverse(void) const {if (m_pHead != NULL) m_pHead->traverse();}
  void traverse(ofstream& ofs) const {if (m_pHead != NULL) m_pHead->traverse(ofs);}
  const LPtrSListNode* getHead(void) const {return m_pHead;}
  BOOL search (const int data) const {if (m_pHead != NULL) return m_pHead->search(data); else return FALSE;}
  long space(void) const { if (m_pHead != NULL) return (sizeof (*this) + m_pHead->space()); else return sizeof(*this);}

// ----------------------
// Attributes
// ----------------------
private:
  LPtrSListNode* m_pHead;
};

#endif // LPTRSLIST_H
```

Code Listing 26: *LPtrSList.cpp*

```
// ****************************************************************
// FILE:            LPtrSList.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Dec. 02, 1997
// DESCRIPTION:  Refer to LPtrSList.h
// ****************************************************************

#include <assert.h>
#include "LPtrSList.h"
#include "LPtrSListNode.h"

// ----------------------------------------------------------
// Function:          LPtrSList()
// Descritpion:     Default constructor. Initialize the head
//                      to be a NULL pointer
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LPtrSList::LPtrSList
  (void)
{
  m_pHead= NULL;
}

// ----------------------------------------------------------
// Function:          LPtrSList()
// Descritpion:     Copy constructor.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LPtrSList::LPtrSList
  (const LPtrSList& rList)
{
  *this= rList;
}


// ----------------------------------------------------------
// Function:          ~LPtrSList()
// Descritpion:     Destructor. Delete the entire binary list.
//                      Free memory allocated.
// Errors Generated: None
// Limitations:
// ----------------------------------------------------------
LPtrSList::~LPtrSList(void)
{
  empty();
}


// ------------------------------------------------------------------
// Function:          insert()
// Descritpion:     Insert data in front of the singly linked list.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------------
void LPtrSList::insert
  (const int data)
{
  if (m_pHead == NULL)
    m_pHead= new LPtrSListNode(data);
  else
    m_pHead= m_pHead->insert(data);
}
```

```
// ------------------------------------------------------------
// Function:        append()
// Descritpion:     Append data at the end of the linked list.
//
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
void LPtrSList::append
  (const int data)
{
  if (m_pHead == NULL)
    m_pHead= new LPtrSListNode(data);
   else
    m_pHead= m_pHead->append(data);
}


// ------------------------------------------------------------
// Function:        remove()
// Descritpion:     Remove data from the singly linked list.
//                  Return the number of nodes that is being
//                  removed from linked list
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
int LPtrSList::remove
  (const int data)
{
  BOOL bDeleteHead;
  int  nCount;

  if (m_pHead != NULL)
  {
    LPtrSListNode* pOldHead= m_pHead;

    m_pHead= m_pHead->remove(data, bDeleteHead, nCount);

    if (bDeleteHead)
    {
      delete pOldHead;
    }
  }

  return nCount;
}

// ------------------------------------------------------------
// Function:        operator=()
// Descritpion:     Copy the entire singly linked list. Allocate
//                  memory as needed.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
LPtrSList& LPtrSList::operator=
  (const LPtrSList& list)
{
  if (m_pHead == NULL)
  {
    if (list.getHead() != NULL)
    {
      m_pHead= new LPtrSListNode(list.getHead()->getSelfData());
      *m_pHead= *list.getHead();
    }
  }
  else
  {
    if (list.getHead() == NULL)
    {
      m_pHead->removeSublist();
      delete m_pHead;
```

```
      m_pHead= NULL;
    }
    else
    {
      *m_pHead= *list.getHead();
    }
  }

  return *this;
}

// ---------------------------------------------------------------
// Function:          empty()
// Descritpion:       Empty the entire singly linked list.
//                    Set the head node to be a NULL pointer.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
void LPtrSList::empty
  (void)
{
  if (m_pHead != NULL)
  {
    m_pHead->removeSublist();
    delete m_pHead;
    m_pHead= NULL;
  }
}
```

Code Listing 27: *LStatistic.h*

```
// ****************************************************************
// FILE:            LStatistic.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Oct. 30. 1997
// DESCRIPTION:  This file contains the declaration of two
//                    classes LCompare and LStatistic. The two
//                    classes are used to generate the statistical
//                    result comparing the null object method and
//                    the null pointer method. Build time, access time
//                    and space required with the two methods are
//                    compared.
// ****************************************************************

#ifndef LSTATISTIC_H
#define LSTATISTIC_H

class LDiv0Exception {};
class LCompare
{
  friend class LStatistic;

// ----------------------------------------
// Constructors, destructors and enum types.
// ----------------------------------------

// ----------
// Methods
// ----------
public:
  void calcRatio(void)
  {
    if (m_nullPtrVal == 0)
    {
      LDiv0Exception eDiv0;

      cout << "divide by 0.\n" << endl;
      throw eDiv0;
    }
    else
      m_ratio= m_nullPtrVal / m_nullObjVal;
  }

// -----------
// Attributes
// -----------
private:
  double  m_nullPtrVal,
          m_nullObjVal,
          m_ratio;
};

class LStatistic
{
// ----------------------------------------
// Constructors, destructors and enum types.
// ----------------------------------------

public:
  enum
  {
    accessTime_E,
    buildTime_E,
    space_E
  };

// -------------
```

```
// Methods
// -------------
public:

  // inline methods
  void setNodeNumber (const unsigned long nodeNum)
  {
    m_nodeNum= nodeNum;
  }

  void setPtrBuildTime (const double nullPtrTime)
  {
    m_buildTime.m_nullPtrVal= nullPtrTime;
  }

  void setObjBuildTime (const double nullObjTime)
  {
    m_buildTime.m_nullObjVal= nullObjTime;
  }

  void setPtrAccessTime (const double nullPtrTime)
  {
    m_accessTime.m_nullPtrVal= nullPtrTime;
  }

  void setObjAccessTime (const double nullObjTime)
  {
    m_accessTime.m_nullObjVal= nullObjTime;
  }

  void setPtrSpace (const double nullPtrSpace)
  {
    m_space.m_nullPtrVal= nullPtrSpace;
  }

  void setObjSpace (const double nullObjSpace)
  {
    m_space.m_nullObjVal= nullObjSpace;
  }

  void calcRatioes    (void)
  {
    m_accessTime.calcRatio();
    m_buildTime.calcRatio();
    m_space.calcRatio();
  }

  double getAccessTimeRatio(void) const
  {
    return m_accessTime.m_ratio;
  }

  double getBuildTimeRatio(void) const
  {
    return m_buildTime.m_ratio;
  }

  double getSpaceRatio(void) const
  {
    return m_space.m_ratio;
  }

// methods defined in the implementation file
  void printReport  (const int rptType) const;
  void printReport  (ofstream& ofs, const int rptType) const;

// -------------
// Attributes
// -------------
private:
```

```
  unsigned long  m_nodeNum;
  LCompare       m_buildTime,
                 m_accessTime,
                 m_space;
};

#endif // LSTATISTIC_H
```

Code Listing 28: *LStatistic.cpp*

```
// ****************************************************************
// FILE:            LStatistic.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Sept. 15, 1997
// DESCRIPTION:  Refer to LStatistic.h
// ****************************************************************


#include <iostream.h>
#include <fstream.h>
#include <assert.h>

#include "LStatistic.h"
#include "LTypes.h"

// ---------------------------------------------------------
// Function:         printReport()
// Descritpion:     Print the statistical results to the
//                      standard output. Access time ratio,
//                      build time ratio and space ratio are
//                      calculated as: the time or space required
//                     with the null pointer method divided by
//                     the time or space required by the null
//                     object method.
//
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
void LStatistic::printReport
  (const int rptType) const
{
  cout << endl;

  switch(rptType)
  {
    case accessTime_E:
      cout << "AccessTimeRatio(" << m_nodeNum << ")= " << m_accessTime.m_ratio << endl;
      break;
    case buildTime_E:
      cout << "BuildTimeRatio(" << m_nodeNum << ")= " << m_buildTime.m_ratio << endl;
      break;
    case space_E:
      cout << "SpaceRatio(" << m_nodeNum << ")= " << m_space.m_ratio << endl;
      break;
    default:
      assert(FALSE);
  }
}


// ---------------------------------------------------------
// Function:         printReport()
// Descritpion:     Print the statistical results to
//                      a file stream. Access time ratio,
//                     build time ratio and space ratio are
//                     calculated as: the time or space required
//                     with the null pointer method divided by
//                     the time or space required by the null
//                     object method.
//
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------
void LStatistic::printReport
  (ofstream& ofs,
   const int rptType) const
```

```
{
  switch(rptType)
  {
    case accessTime_E:
      ofs << "AccessTimeRatio(" << m_nodeNum << ")= " << m_accessTime.m_ratio << endl;
      break;
    case buildTime_E:
      ofs << "BuildTimeRatio(" << m_nodeNum << ")= " << m_buildTime.m_ratio << endl;
      break;
    case space_E:
      ofs << "SpaceRatio(" << m_nodeNum << ")= " << m_space.m_ratio << endl;
      break;
    default:
      assert(FALSE);
  }
}
```

Code Listing 29: *LRandomGen.h*

```
// ****************************************************************
// FILE:            LRandomGen.h
// AUTHOR:      Zhilin Li
// CREATION DATE: Oct. 17, 1997
// DESCRIPTION:  This file implements a class LRandomGen which
//                    can be used to generate a random positive
//                    integer number.
// ****************************************************************

#ifndef LRANDOMGEN_H
#define LRANDOMGEN_H

#include <stddef.h>
#include <time.h>
#include <limits.h>

class LRandomGen
{
// ----------------------------
// Constructors and destructors
// ----------------------------

public:
  LRandomGen(unsigned long seed= (unsigned long) time(NULL)) : m_seed(seed) {}

// --------------------------
// Methods
// --------------------------
  void setSeed(unsigned long newSeed= (unsigned long) time(NULL)) {m_seed= newSeed;}

// ---------------
// operators
// ---------------

  int operator() (int lim= INT_MAX)
  {
    m_seed= m_seed * 5709421UL + 1UL;
    return ((int) (m_seed % lim));
  }

// -----------------
// Attributes
// -----------------
private:
  unsigned long m_seed;
};

#endif
```

Code Listing 30: *LTypes.h*

```
// File LTypes.h

#ifndef LTYPES_H
#define LTYPES_H

#define FALSE 0
#define TRUE  1

typedef int BOOL;


#endif /* LTYPES_H */
```

Code Listing 31: *main.cpp*

```
// ************************************************************
// FILE:        main.cpp
// AUTHOR:      Zhilin Li
// CREATION DATE: Sept. 15, 1997
// DESCRIPTION:  This file contains the main() function and other
//              test functions.
// ************************************************************

// ----------------
// Include files
// ----------------

#include <sys/types.h>
#include <sys/timeb.h>  // for time functions
#include <fstream.h>
#include <stdio.h>
#include <assert.h>
#include <conio.h>
#include <string.h>

#include "LBTree.h"
#include "LPtrBTree.h"

#include "LSList.h"
#include "LPtrSList.h"

#include "LTmplBTree.h"
#include "LTmplPtrBTree.h"

#include "LTmplStack.h"
#include "LTmplPtrStack.h"


#include "LNull.h"
#include "LRandomGen.h"
#include "LStatistic.h"

// -----------------
// macro definitions
// -----------------

#define  STA_FOR_LOOP(j)  for (int j= largeStartNode_G; j <= largeEndNode_G; ((j >= 300000) ? (j+= 100000) : (j+=
increment_G)))

// -----------------
// Global variables
// -----------------

static const int     arraySize= 20;
static int           staNum_G= 0;
static LStatistic    staBTreeResult_G[arraySize];
static LStatistic    staSListResult_G[arraySize];
static LStatistic    staDblTmplBTreeResult_G[arraySize];
static LStatistic    staIntTmplBTreeResult_G[arraySize];
static LStatistic    staAVLBTreeResult_G[arraySize];

// Used to generate a set of large binary trees or linked list with node numbers between
// largeStartNode and largeEndNode. The set of binary trees are used
// to compare the time and space efficiency between the null object method
// and the null pointer method.
static int largeStartNode_G; // 10000
static int largeEndNode_G;   // 400000
static int increment_G;      // 20000


// ----------------
```

```
// Local variables
// ----------------

enum
{
  binaryTree_E,
  singleLinkedList_E,
  intTmplBinaryTree_E,
  dblTmplBinaryTree_E,
  AVLBinaryTree_E,
};

enum
{
  ptr_E,
  obj_E,
};

// Seed that is used as the start for generating a series
// of random numbers.
static const int seed_S= 919;

// testTreeFile_S is the test result file for LBTree.
// testPtrTreeFile_S is the test result file for LPtrBTree.
static const char*  testTreeFile_S= "TestTree.txt";
static const char*  testPtrTreeFile_S= "TestPtrTree.txt";

// testIntTmplTreeFile_S is the test result file for LTmplBTree<int>.
// testIntTmplPtrTreeFile_S is the test result file for LTmplPtrBTree<int>.
static const char*  testIntTmplTreeFile_S= "TestIntTmplTree.txt";
static const char*  testIntTmplPtrTreeFile_S= "TestIntTmplPtrTree.txt";

// testDblTmplTreeFile_S is the test result file for LTmplBTree<double>.
// testDblTmplPtrTreeFile_S is the test result file for LTmplPtrBTree<double>.
static const char*  testDblTmplTreeFile_S= "TestDblTmplTree.txt";
static const char*  testDblTmplPtrTreeFile_S= "TestDblTmplPtrTree.txt";

// testAVLTreeFile_S is the test result file for LAVLBTree.
// testPtrAVLTreeFile_S is the test result file for LPtrAVLBTree.
static const char*  testAVLTreeFile_S= "TestAVLTree.txt";
static const char*  testPtrAVLTreeFile_S= "TestPtrAVLTree.txt";

// testSListFile_S is the test result file for LSList.
// testPtrSListFile_S is the test result file for LPtrSList.
static const char*  testSListFile_S= "TestSList.txt";
static const char*  testPtrSListFile_S= "TestPtrSList.txt";


// statisticFile_S is the result for comparing the two
// methods: null pointer method and null object method.
static const char*  statisticFile_S= "TestStatistic.txt";

// Used to generate a small binary tree with nodeNum_S nodes.
// Each node is an integer less than maxTestNum_S.
// The small binary tree is used to test basic functionality
// of the binary tree.
static const int maxTestNum_S= 10000;
static const int nodeNum_S= 10;

// Used to test insert, delete some casually selected numbers.
static const int num1_S= 12345;
static const int num2_S= 4321;
static const int num3_S= 15;
static const int num4_S= 8876;
static const int num5_S= 5034;

// a small utility to print out a new test header.
static char strBorder_S[]= "////////////////////////////////////////////////////////////////////////////////";
inline static void newTest(ofstream& ofs, const char* s) {ofs << endl << endl << strBorder_S << endl << s << endl <<
strBorder_S << endl << endl;}
```

```
inline static void setTestNum(int start, int end, int increment)
{
  largeStartNode_G= start;
  largeEndNode_G= end;
  increment_G= increment;

  staNum_G= 0;

  STA_FOR_LOOP(j)
  {
    staNum_G++;
  }
}

// ------------------------------------------------------
// initilization of the public static member variables
// of the template class LTmplFullBTreeNode.
// ------------------------------------------------------

LTmplNullBTreeNode<int> LTmplFullBTreeNode<int>::m_nullNode_S;
BOOL LTmplFullBTreeNode<int>::m_bCountNullNodeSpace_S= FALSE;

LTmplNullBTreeNode<double> LTmplFullBTreeNode<double>::m_nullNode_S;
BOOL LTmplFullBTreeNode<double>::m_bCountNullNodeSpace_S= FALSE;

LTmplNullStackNode<int> LTmplFullStackNode<int>::m_nullNode_S;

// -------------------------
// Function Prototypes
// -------------------------

template <class TYPE>
void generalTest (TYPE &testType, ofstream& testFile, LStatistic* staArray, int methodType, BOOL isLinkedList= FALSE);

template <class TYPE>
void slistTest (TYPE &testType, ofstream& testFile);

template <class TYPE>
void AVLBTreeTest(TYPE &testType, ofstream &testFile);

// print out the statistical result
void printStaReport(ofstream& ofs, const int testType);


// ---------------------------------------------------------------
// Function:        main
// Descritpion:     main() function for the test program.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
int main(void)
{
  ofstream  staFile(statisticFile_S);
  ofstream testFile;

  setTestNum(10000, 400000, 20000);

  // ---------------------------------
  // Print out the statistical result.
  // ---------------------------------

  staFile << endl;
  staFile << "NOTE THAT:\n";
  staFile << "          time or space required using null pointer method\n";
  staFile << "Ratioes =  ----------------------------------------------\n";
  staFile << "          time or space required using null object method\n";
  staFile << endl << endl;

  // ----------------
  // Test binary tree.
```

```
// ----------------

// test null object method
cout << "test binary tree -- null object" << endl;

testFile.open(testTreeFile_S, ios::out);
LBTree objBTree;

testFile << "\t================================================" << endl;
testFile << "\t Test Binary Tree Built With Null Node Object" << endl;
testFile << "\t================================================" << endl;

generalTest(objBTree, testFile, staBTreeResult_G, obj_E);

testFile.close();

// test null pointer method.

cout << "test binary tree - null pointer" << endl;

testFile.open(testPtrTreeFile_S, ios::out);
LPtrBTree ptrBTree;

testFile << "\t================================================" << endl;
testFile << "\t Test Binary Tree Built With NULL pointers" << endl;
testFile << "\t================================================" << endl;

generalTest(ptrBTree, testFile, staBTreeResult_G, ptr_E);

testFile.close();

// print the statistical result
printStaReport(staFile, binaryTree_E);

// ---------------------------------------
// Test template binary tree of double.
// ---------------------------------------

// test null object method
cout << "test template binary tree <double> -- null object" << endl;

testFile.open(testDblTmplTreeFile_S, ios::out);
LTmplBTree<double> objDblTmplBTree;

testFile << "\t=====================================================================" <<
endl;
testFile << "\t Test Template Binary Tree <double> Built With Null Node Objects" << endl;
testFile << "\t=====================================================================" <<
endl;

generalTest(objDblTmplBTree, testFile, staDblTmplBTreeResult_G, obj_E);

testFile.close();

// test null pointer method.

cout << "test template binary tree <double> -- null pointer" << endl;

testFile.open(testDblTmplPtrTreeFile_S, ios::out);
LTmplPtrBTree<double> ptrDblTmplBTree;

testFile << "\t=====================================================================" << endl;
testFile << "\t Test Template Binary Tree <double> Built With NULL pointers" << endl;
testFile << "\t=====================================================================" << endl;

generalTest(ptrDblTmplBTree, testFile, staDblTmplBTreeResult_G, ptr_E);

testFile.close();

// print the statistical result
```

```
    printStaReport(staFile, dblTmplBinaryTree_E);

  // ---------------------------------------
    // Test template binary tree of int.
    // ---------------------------------------

    // test null object method
    cout << "test template binary tree <int> -- null object" << endl;

    testFile.open(testIntTmplTreeFile_S, ios::out);
    LTmplBTree<int> objIntTmplBTree;

    testFile << "\t============================================================" << endl;
    testFile << "\t Test Template Binary Tree <int> Built With Null Node Objects" << endl;
    testFile << "\t============================================================" << endl;

    generalTest(objIntTmplBTree, testFile, staIntTmplBTreeResult_G, obj_E);

    testFile.close();

    // test null pointer method.

    cout << "test template binary tree <int> -- null pointer" << endl;

    testFile.open(testIntTmplPtrTreeFile_S, ios::out);
    LTmplPtrBTree<int> ptrIntTmplBTree;

    testFile << "\t============================================================" << endl;
    testFile << "\t Test Template Binary Tree <int> Built With NULL pointers" << endl;
    testFile << "\t============================================================" << endl;

    generalTest(ptrIntTmplBTree, testFile, staIntTmplBTreeResult_G, ptr_E);

    testFile.close();

    // print the statistical result
    printStaReport(staFile, intTmplBinaryTree_E);

  // ---------------------------------
    // Test AVL binary tree.
    // ---------------------------------

    // test null object method
    cout << "test AVL binary tree -- null object" << endl;

    testFile.open(testAVLTreeFile_S, ios::out);
    LAVLBTree objAVLBTree;

    testFile << "\t================================================" << endl;
    testFile << "\t Test AVL Binary Tree Built With Null Node Object" << endl;
    testFile << "\t================================================" << endl;

    AVLBTreeTest(objAVLBTree, testFile);
    generalTest(objAVLBTree, testFile, staAVLBTreeResult_G, obj_E);

    testFile.close();

    // test null pointer method.

    cout << "test AVL binary tree - null pointer" << endl;

    testFile.open(testPtrAVLTreeFile_S, ios::out);
    LPtrAVLBTree ptrAVLBTree;

    testFile << "\t================================================" << endl;
    testFile << "\t Test AVL Binary Tree Built With NULL pointers" << endl;
    testFile << "\t================================================" << endl;

    AVLBTreeTest(ptrAVLBTree, testFile);
    generalTest(ptrAVLBTree, testFile, staAVLBTreeResult_G, ptr_E);
```

```
  testFile.close();

  // print the statistical result
  printStaReport(staFile, AVLBinaryTree_E);

// ------------------------
  // Test singly linked list
  // ------------------------

  setTestNum(10000, 100000, 10000);

  // test null object method
  cout << "test singly linked list -- null object" << endl;

  testFile.open(testSListFile_S, ios::out);
  LSList objSList;

  testFile << "\t=====================================================" << endl;
  testFile << "\t Test Singly linked list Built With Null Node Object" << endl;
  testFile << "\t=====================================================" << endl;

  slistTest(objSList, testFile);
  generalTest(objSList, testFile, staSListResult_G, obj_E,TRUE);

  testFile.close();

  // test null pointer method.
  cout << "test singly linked list - null pointer" << endl;

  testFile.open(testPtrSListFile_S, ios::out);
  LPtrSList ptrSList;

  testFile << "\t=====================================================" << endl;
  testFile << "\t Test Singly linked list Built With NULL pointers" << endl;
  testFile << "\t=====================================================" << endl;

  slistTest(ptrSList, testFile);
  generalTest(ptrSList, testFile, staSListResult_G, ptr_E, TRUE);

  testFile.close();

  // print the statistical result
  printStaReport(staFile, singleLinkedList_E);

  return 0;
}

// ------------------------------------------------------------
// Function:       printStaReport
// Descritpion:    Print out the statistical report.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
void printStaReport
  ( ofstream& staFile,
    const int testType)
{

  LStatistic* pStaArray;
  int i;

  switch (testType)
  {
    case binaryTree_E:

      // ------------------------------------------
      // Print out the statistical results
      // for implementing the binary tree.
      // ------------------------------------------
```

```
    staFile << "=========================================================== \n";
    staFile << " Compare the two methods for implementing a binary tree \n";
    staFile << "===========================================================\n\n" ;

    pStaArray= staBTreeResult_G;

    break;

  case singleLinkedList_E:

    // -----------------------------------------
    // Print out the statistical results
    // for implementing the binary tree.
    // -----------------------------------------

    staFile << "================================================================= \n";
    staFile << " Compare the two methods for implementing a singly linked list \n";
    staFile << "=================================================================\n\n" ;

    pStaArray= staSListResult_G;

    break;

  case dblTmplBinaryTree_E:

    // -----------------------------------------------------
    // Print out the statistical results
    // for implementing the template binary tree of double.
    // -----------------------------------------------------

    staFile << "================================================================= \n";
    staFile << " Compare the two methods for implementing a template binary tree \n";
    staFile << " Data type is: double\n";
    staFile << "=================================================================\n\n" ;

    pStaArray= staDblTmplBTreeResult_G;

    break;

  case intTmplBinaryTree_E:

    // -----------------------------------------------------
    // Print out the statistical results
    // for implementing the template binary tree of int.
    // -----------------------------------------------------

    staFile << "================================================================= \n";
    staFile << " Compare the two methods for implementing a template binary tree \n";
    staFile << " Data type is: integer\n";
    staFile << "=================================================================\n\n" ;

    pStaArray= staIntTmplBTreeResult_G;

    break;

  case AVLBinaryTree_E:

    // -----------------------------------------
    // Print out the statistical results
    // for implementing the AVL binary tree.
    // -----------------------------------------

    staFile << "================================================================= \n";
    staFile << " Compare the two methods for implementing an AVL binary tree   \n";
    staFile << "=================================================================\n\n" ;

    pStaArray= staAVLBTreeResult_G;

    break;
```

```
    default:
      assert(FALSE);
      break;
  }

  double minAccessTimeRatio,
         maxAccessTimeRatio,
         aveAccessTimeRatio,
         sumAccessTimeRatio= 0.
        minBuildTimeRatio,
        maxBuildTimeRatio,
        aveBuildTimeRatio,
        sumBuildTimeRatio= 0.
        minSpaceRatio,
        maxSpaceRatio,
        aveSpaceRatio,
       sumSpaceRatio= 0;

  // ------------------------------------------
  // Calculate the access time, the build time
  // and the space ratioes for the two methods.
  // ------------------------------------------

  for ( i= 0; i < staNum_G; i++)
  {
    pStaArray[i].calcRatioes();

    if (i == 0)
    {
      minAccessTimeRatio= maxAccessTimeRatio= sumAccessTimeRatio= pStaArray[i].getAccessTimeRatio();
      minBuildTimeRatio= maxBuildTimeRatio= sumBuildTimeRatio= pStaArray[i].getBuildTimeRatio();
      minSpaceRatio= maxSpaceRatio= sumSpaceRatio= pStaArray[i].getSpaceRatio();
    }
    else
    {
     minAccessTimeRatio= (minAccessTimeRatio <= pStaArray[i].getAccessTimeRatio())? minAccessTimeRatio :
pStaArray[i].getAccessTimeRatio();
      maxAccessTimeRatio= (maxAccessTimeRatio >= pStaArray[i].getAccessTimeRatio())? maxAccessTimeRatio :
pStaArray[i].getAccessTimeRatio();
      sumAccessTimeRatio+= pStaArray[i].getAccessTimeRatio();

      minBuildTimeRatio= (minBuildTimeRatio <= pStaArray[i].getBuildTimeRatio())? minBuildTimeRatio :
pStaArray[i].getBuildTimeRatio();
      maxBuildTimeRatio= (maxBuildTimeRatio >= pStaArray[i].getBuildTimeRatio())? maxBuildTimeRatio :
pStaArray[i].getBuildTimeRatio();
      sumBuildTimeRatio+= pStaArray[i].getBuildTimeRatio();

      minSpaceRatio= (minSpaceRatio <= pStaArray[i].getSpaceRatio())? minSpaceRatio : pStaArray[i].getSpaceRatio();
      maxSpaceRatio= (maxSpaceRatio >= pStaArray[i].getSpaceRatio())? maxSpaceRatio : pStaArray[i].getSpaceRatio();
      sumSpaceRatio+= pStaArray[i].getSpaceRatio();
    }
  }

  aveAccessTimeRatio= sumAccessTimeRatio / staNum_G;
  aveBuildTimeRatio= sumBuildTimeRatio / staNum_G;
  aveSpaceRatio= sumSpaceRatio / staNum_G;

  // ------------------------------------------
  // print out the result about the build time
  // ------------------------------------------

  staFile << "Build Time Ratio:\n";
  staFile << "-----------------\n";

  for ( i= 0; i < staNum_G; i++)
  {
    pStaArray[i].printReport(staFile, LStatistic::buildTime_E);
  }
```

```
  staFile << "BuildTimeRatio(min)= " << minBuildTimeRatio << endl;
  staFile << "BuildTimeRatio(max)= " << maxBuildTimeRatio << endl;
  staFile << "BuildTimeRatio(ave)= " << aveBuildTimeRatio << endl;
  staFile << endl;

  // ----------------------------------------
  // print out the result about the access time
  // ----------------------------------------

  staFile << "Access Time Ratio: \n";
  staFile << "-----------------\n";

  for ( i= 0; i < staNum_G; i++)
  {
    pStaArray[i].printReport(staFile, LStatistic::accessTime_E);
  }

  staFile << "AccessTimeRatio(min)= " << minAccessTimeRatio << endl;
  staFile << "AccessTimeRatio(max)= " << maxAccessTimeRatio << endl;
  staFile << "AccessTimeRatio(ave)= " << aveAccessTimeRatio << endl;
  staFile << endl;

  // ----------------------------------------
  // print out the result about the space
  // ----------------------------------------

  staFile << "Space Ratio:\n";
  staFile << "------------\n";

  for ( i= 0; i < staNum_G; i++)
  {
    pStaArray[i].printReport(staFile, LStatistic::space_E);
  }

  staFile << "SpaceRatio(min)= " << minSpaceRatio << endl;
  staFile << "SpaceRatio(max)= " << maxSpaceRatio << endl;
  staFile << "SpaceRatio(ave)= " << aveSpaceRatio << endl;
  staFile << endl << endl;
}

// ---------------------------------------------------------------
// Function:       generalTest
// Descritpion:    Template function for testing different
//                 implementation of the trees and linkes list.
// Errors Generated: None
// Limitations:
// ---------------------------------------------------------------
template <class TYPE>
void generalTest (TYPE &testType, ofstream &testFile, LStatistic* staArray, int methodType, BOOL isLinkedList)
{
  char      strBuff[256];

  TYPE      cpType,
            largeType;
  LRandomGen  rg(seed_S);
  int       i;

  // ----------------------------------------
  // Generate nodeNum_S random numbers and
  // insert the numbers into data structure.
  // ----------------------------------------

  for (i= 0; i < nodeNum_S; i++)
  {
    testType.insert(rg(maxTestNum_S));
  }

  sprintf(strBuff, "// Here is a data structure with %d random numbers:", nodeNum_S);
  newTest(testFile, strBuff);
  testType.traverse(testFile);
```

```
// ------------------------------
// Test operator =().
// ------------------------------

cpType= testType;
newTest(testFile, "// The following data structure is a copy of the above one:");
cpType.traverse(testFile);

// ------------------------------------------
// Insert num1_S, num2_S, etc.
// ------------------------------------------

testType.insert(num1_S);
testType.insert(num2_S);
testType.insert(num3_S);
testType.insert(num4_S);
testType.insert(num5_S);

sprintf(strBuff, "// After inserting %ld, %ld, %ld, %ld, %ld:",
      num1_S, num2_S, num3_S, num4_S, num5_S);

newTest(testFile, strBuff);
testType.traverse(testFile);

// ---------------------------------
// Test search
// ---------------------------------

newTest(testFile, "// Search for the newly inserted numbers");

if (testType.search(num1_S))
{
  testFile << "Find number " << num1_S << endl;
}
else
{
  testFile << "Cannot find number " << num1_S << endl;
}

if (testType.search(num2_S))
{
  testFile << "Find number " << num2_S << endl;
}
else
{
  testFile << "Cannot find number " << num2_S << endl;
}

if (testType.search(num3_S))
{
  testFile << "Find number " << num3_S << endl;
}
else
{
  testFile << "Cannot find number " << num3_S << endl;
}

if (testType.search(num4_S))
{
  testFile << "Find number " << num4_S << endl;
}
else
{
  testFile << "Cannot find number " << num4_S << endl;
}

if (testType.search(num5_S))
{
  testFile << "Find number " << num5_S << endl;
```

```
}
else
{
  testFile << "Cannot find number " << num5_S << endl;
}

// ------------------------------------------------
// Try to search for a random generated number.
// ------------------------------------------------
const int randNum= rg(maxTestNum_S);

if (testType.search(randNum))
{
  testFile << "Find random number " << randNum << endl;
}
else
{
  testFile << "Cannot find random number " << randNum << endl;
}

// ---------------------------------------
// Remove the first three test numbers
// ---------------------------------------

newTest(testFile, "// Remove test");

testType.remove(num1_S);
testType.remove(num2_S);
testType.remove(num3_S);

testFile << "===After removing ";
testFile << num1_S << ", " << num2_S << ", " << num3_S << ". \n\n";

testType.traverse(testFile);

// ----------------------------------
// Remove the last two test numbers.
// ----------------------------------

testType.remove(num4_S);
testType.remove(num5_S);

testFile << endl << "===After removing ";
testFile << num4_S << ", " << num5_S << ". \n\n";

testType.traverse(testFile);

// ------------------------------------------------
// Remove a random genarated number.
// ------------------------------------------------

newTest(testFile, "// Remove a random number from the above structure.");
if (testType.remove(randNum))
  testFile << "Successully remove " << randNum << endl;
else
  testFile << "Cannot remove " << randNum << endl;


// ----------------------------
// Empty struture.
// ----------------------------

newTest(testFile, "// After empty(), the above data structure become:" );
testType.empty();
testType.traverse(testFile);

// ------------------------------------------------------
// Measure build time, access time and space requirements
// to build a large tree or linked list.
// ------------------------------------------------------
```

```
struct _timeb startTime, endTime;
double elapsedTime;

newTest(testFile, "// Measure time needed to build and access the data structure.");

int testNum= 0;

STA_FOR_LOOP(j)
{
  cout << j << endl;

  // ---------------------------------------------------
  // Store the node number to the statistical result array.
  // ---------------------------------------------------

  staArray[testNum].setNodeNumber(j);

  testFile << "====== Data structure with " << j << " nodes ======" << endl;

  // ---------------------------------
  // Time used to build a j-node tree
  // ---------------------------------

  _ftime(&startTime);
  for (i= 0; i < j; i++)
  {
    largeType.insert(rg());
  }
  _ftime(&endTime);

  elapsedTime= (endTime.time - startTime.time) * 1000 + endTime.millitm - startTime.millitm;

  testFile << "Time to build the data structure: " << elapsedTime << " milliseconds" << endl;

  // ---------------------------------------------------------------
  // Store the build time using the null object method into the
  // statistical result array.
  // ---------------------------------------------------------------

  if (methodType == obj_E)
    staArray[testNum].setObjBuildTime(elapsedTime);
  else if (methodType == ptr_E)
    staArray[testNum].setPtrBuildTime(elapsedTime);
  else
    assert(0);

  // ---------------------------------------
  // Time that is needed to access the data
  // structure for j times.
  // ---------------------------------------

  if (isLinkedList)
  {
    _ftime(&startTime);
    for (i= 0; i < 500; i++)
    {
      largeType.search(rg());
    }
    _ftime(&endTime);
  }
  else
  {
    _ftime(&startTime);
    for (i= 0; i < j; i++)
    {
      largeType.search(rg());
    }
    _ftime(&endTime);
  }
```

```
    elapsedTime= (endTime.time - startTime.time) * 1000 + endTime.millitm - startTime.millitm;
    testFile << "Time to access the data structure for " << j << " times: " << elapsedTime << " milliseconds" << endl;

    // ------------------------------------------------------------
    // Store the access time using the null object method into the
    // statistical result array.
    // ------------------------------------------------------------

    if (methodType == obj_E)
      staArray[testNum].setObjAccessTime(elapsedTime);
    else if (methodType == ptr_E)
      staArray[testNum].setPtrAccessTime(elapsedTime);
    else
      assert(0);

    // ---------------------------------
    // Calculate the space required by
    // the tree.
    // ---------------------------------

    unsigned long treeSpace= largeType.space();
    testFile << "Space required to build the data structure: " << treeSpace << " bytes" << endl << endl;

    // ------------------------------------------------------------
    // Store the space required using the null object method into the
    // statistical result array.
    // ------------------------------------------------------------

    if (methodType == obj_E)
      staArray[testNum].setObjSpace(treeSpace);
    else if (methodType== ptr_E)
      staArray[testNum].setPtrSpace(treeSpace);
    else
      assert(0);

    // ---------------------------
    // Empty the large tree.
    // ---------------------------

    largeType.empty();
    testNum++;
  }
}



// ------------------------------------------------------------
// Function:       slistTest
// Descritpion:    Do part of the basic functional test for
//                 singly linked list.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------
template <class TYPE>
void slistTest (TYPE &testType, ofstream &testFile)
{
  int       i;

  // ---------------------------------------------------
  // Test insert(), append() and remove()
  // ---------------------------------------------------

  for (i= 1; i <= 10; i++)
  {
    testType.insert(i);
  }

  newTest(testFile, "// Test insert(), append() and remove()");
  testFile << endl << "After inserting numbers 1 to 10: " << endl;
```

```
  testType.traverse(testFile);

  for (i= 1; i <= 10; i++)
  {
    testType.append(i);
  }

  testFile << endl << endl <<  "After appending numbers 1 to 10: " << endl;
  testType.traverse(testFile);

  int nRemoved;

  nRemoved= testType.remove(1);
  testFile << endl << "After removing 1:" << endl;
  testType.traverse(testFile);
  testFile << endl << "Number of nodes removed: " << nRemoved << endl;

  nRemoved= testType.remove(2);
  testFile << endl << "After removing 2:" << endl;
  testType.traverse(testFile);
  testFile << endl << "Number of nodes removed: " << nRemoved << endl;

  nRemoved= testType.remove(11);
  testFile << endl << "After removing 11:" << endl;
  testType.traverse(testFile);
  testFile << endl << "Number of nodes removed: " << nRemoved << endl;

  testType.empty();

  // -------------------------------
  // Insert five 2 and remove 2.
  // -------------------------------

  for (i= 0; i< 5; i++)
  {
    testType.insert(2);
  }

  newTest(testFile, "// Insert five 2s and remove 2:");

  testFile << endl << "After insert five 2s:" << endl;
  testType.traverse(testFile);

  nRemoved= testType.remove(2);
  testFile << endl << "After removing 2:" << endl;
  testType.traverse(testFile);
  testFile << endl << "Number of nodes removed: " << nRemoved << endl;

  testType.empty();
}


// ------------------------------------------------------------------
// Function:        AVLBTreeTest
// Descritpion:     Do part of the functional test for AVL binary
//                  tree.
// Errors Generated: None
// Limitations:
// ------------------------------------------------------------------
template <class TYPE>
void AVLBTreeTest(TYPE &testType, ofstream &testFile)
{
  char       strBuff[256];
  int        i;
  LRandomGen  rg(seed_S);

  // ------------------------------------------------
  // Build a tree with numbers 1 to 100.
  // ------------------------------------------------
```

```
  for (i= 1; i <= 100 ; i++)
  {
    testType.insert(i);
  }

  sprintf(strBuff, "// Here is a height balance tree with numbers 1 to %d (right rotation):", i-1);
  newTest(testFile, strBuff);
  testType.traverse(testFile);

  sprintf(strBuff, "After removing ");
  char tmp[10];

  for (i= 200; i >= 1; i--)
  {
    int x= rg(100);
    testType.remove(x);
    sprintf(tmp, "%d, ", x);
    strcat(strBuff, tmp);
    if ((i % 20 == 0) && (i != 200))
    {
      newTest(testFile, strBuff);
      testType.traverse(testFile);
      sprintf(strBuff, "After removing ");
    }
  }

  // ------------------------------------
  // Build a tree with numbers 30 to 1
  // ------------------------------------

  for (i= 30; i >= 1; i--)
  {
    testType.insert(i);
  }

  sprintf(strBuff, "// Here is a height balance tree with numbers 30 to 1 (left rotation):");
  newTest(testFile, strBuff);
  testType.traverse(testFile);
  testType.empty();

  // ------------------------------------------------
  // Select a group of number to test double rotation.
  // 30, 10, 50, 55, 51, 5, 9, 52, 53
  // ------------------------------------------------

  testType.insert(30);
  testType.insert(10);
  testType.insert(50);
  testType.insert(55);
  testType.insert(51);
  testType.insert(5);
  testType.insert(9);
  testType.insert(52);
  testType.insert(53);

  sprintf(strBuff, "// Here is a height balance tree with double rotation");
  newTest(testFile, strBuff);
  testType.traverse(testFile);
  testType.empty();

}
```

Test Result 1: *TestTree.txt*

```
=============================================
 Test Binary Tree Built With Null Node Object
=============================================

////////////////////////////////////////////////////////////////////
// Here is a data structure with 10 random numbers:
////////////////////////////////////////////////////////////////////

        NIL     <======          35      ======>         NIL
         35     <======         604      ======>        1213
        NIL     <======         810      ======>         NIL
        810     <======        1213      ======>        6152
        NIL     <======        3956      ======>         NIL
       3956     <======        6102      ======>         NIL
       6102     <======        6152      ======>        9609
        NIL     <======        6383      ======>        9221
        NIL     <======        9221      ======>         NIL
       6383     <======        9609      ======>         NIL

////////////////////////////////////////////////////////////////////
// The following data structure is a copy of the above one:
////////////////////////////////////////////////////////////////////

        NIL     <======          35      ======>         NIL
         35     <======         604      ======>        1213
        NIL     <======         810      ======>         NIL
        810     <======        1213      ======>        6152
        NIL     <======        3956      ======>         NIL
       3956     <======        6102      ======>         NIL
       6102     <======        6152      ======>        9609
        NIL     <======        6383      ======>        9221
        NIL     <======        9221      ======>         NIL
       6383     <======        9609      ======>         NIL

////////////////////////////////////////////////////////////////////
// After inserting 12345, 4321, 15, 8876, 5034:
////////////////////////////////////////////////////////////////////

        NIL     <======          15      ======>         NIL
         15     <======          35      ======>         NIL
         35     <======         604      ======>        1213
        NIL     <======         810      ======>         NIL
        810     <======        1213      ======>        6152
        NIL     <======        3956      ======>        4321
        NIL     <======        4321      ======>        5034
        NIL     <======        5034      ======>         NIL
       3956     <======        6102      ======>         NIL
       6102     <======        6152      ======>        9609
        NIL     <======        6383      ======>        9221
        NIL     <======        8876      ======>         NIL
       8876     <======        9221      ======>         NIL
       6383     <======        9609      ======>       12345
        NIL     <======       12345      ======>         NIL

////////////////////////////////////////////////////////////////////
// Search for the newly inserted numbers
////////////////////////////////////////////////////////////////////

Find number 12345
Find number 4321
Find number 15
Find number 8876
Find number 5034
```

```
Cannot find random number 7762


////////////////////////////////////////////////////////////////////////
// Remove test
////////////////////////////////////////////////////////////////////////

===After removing 12345, 4321, 15,

         NIL     <======          35     ======>         NIL
          35     <======         604     ======>        1213
         NIL     <======         810     ======>         NIL
         810     <======        1213     ======>        6152
         NIL     <======        3956     ======>        5034
         NIL     <======        5034     ======>         NIL
        3956     <======        6102     ======>         NIL
        6102     <======        6152     ======>        9609
         NIL     <======        6383     ======>        9221
         NIL     <======        8876     ======>         NIL
        8876     <======        9221     ======>         NIL
        6383     <======        9609     ======>         NIL

===After removing 8876, 5034,

         NIL     <======          35     ======>         NIL
          35     <======         604     ======>        1213
         NIL     <======         810     ======>         NIL
         810     <======        1213     ======>        6152
         NIL     <======        3956     ======>         NIL
        3956     <======        6102     ======>         NIL
        6102     <======        6152     ======>        9609
         NIL     <======        6383     ======>        9221
         NIL     <======        9221     ======>         NIL
        6383     <======        9609     ======>         NIL


////////////////////////////////////////////////////////////////////////
// Remove a random number from the above structure.
////////////////////////////////////////////////////////////////////////

Cannot remove 7762


////////////////////////////////////////////////////////////////////////
// After empty(), the above data structure become:
////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////
// Measure time needed to build and access the data structure.
////////////////////////////////////////////////////////////////////////

====== Data structure with 10000 nodes ======
Time to build the data structure: 120 milliseconds
Time to access the data structure for 10000 times: 90 milliseconds
Space required to build the data structure: 160012 bytes

====== Data structure with 30000 nodes ======
Time to build the data structure: 371 milliseconds
Time to access the data structure for 30000 times: 330 milliseconds
Space required to build the data structure: 480008 bytes

====== Data structure with 50000 nodes ======
Time to build the data structure: 640 milliseconds
Time to access the data structure for 50000 times: 621 milliseconds
Space required to build the data structure: 800008 bytes

====== Data structure with 70000 nodes ======
Time to build the data structure: 941 milliseconds
Time to access the data structure for 70000 times: 921 milliseconds
```

```
Space required to build the data structure: 1120008 bytes

====== Data structure with 90000 nodes ======
Time to build the data structure: 1462 milliseconds
Time to access the data structure for 90000 times: 1192 milliseconds
Space required to build the data structure: 1439992 bytes

====== Data structure with 110000 nodes ======
Time to build the data structure: 1633 milliseconds
Time to access the data structure for 110000 times: 1622 milliseconds
Space required to build the data structure: 1760008 bytes

====== Data structure with 130000 nodes ======
Time to build the data structure: 1983 milliseconds
Time to access the data structure for 130000 times: 1922 milliseconds
Space required to build the data structure: 2079976 bytes

====== Data structure with 150000 nodes ======
Time to build the data structure: 2384 milliseconds
Time to access the data structure for 150000 times: 2333 milliseconds
Space required to build the data structure: 2399960 bytes

====== Data structure with 170000 nodes ======
Time to build the data structure: 2774 milliseconds
Time to access the data structure for 170000 times: 2774 milliseconds
Space required to build the data structure: 2719960 bytes

====== Data structure with 190000 nodes ======
Time to build the data structure: 3025 milliseconds
Time to access the data structure for 190000 times: 2954 milliseconds
Space required to build the data structure: 3039864 bytes

====== Data structure with 210000 nodes ======
Time to build the data structure: 3605 milliseconds
Time to access the data structure for 210000 times: 3345 milliseconds
Space required to build the data structure: 3359896 bytes

====== Data structure with 230000 nodes ======
Time to build the data structure: 3825 milliseconds
Time to access the data structure for 230000 times: 3726 milliseconds
Space required to build the data structure: 3679960 bytes

====== Data structure with 250000 nodes ======
Time to build the data structure: 4276 milliseconds
Time to access the data structure for 250000 times: 4136 milliseconds
Space required to build the data structure: 3999928 bytes

====== Data structure with 270000 nodes ======
Time to build the data structure: 4747 milliseconds
Time to access the data structure for 270000 times: 4577 milliseconds
Space required to build the data structure: 4319896 bytes

====== Data structure with 290000 nodes ======
Time to build the data structure: 4967 milliseconds
Time to access the data structure for 290000 times: 4757 milliseconds
Space required to build the data structure: 4639800 bytes

====== Data structure with 310000 nodes ======
Time to build the data structure: 5398 milliseconds
Time to access the data structure for 310000 times: 5217 milliseconds
Space required to build the data structure: 4959832 bytes
```

Test Result 2: *TestPtrTree.txt*

```
================================================
 Test Binary Tree Built With NULL pointers
================================================

////////////////////////////////////////////////////////////////////////
// Here is a data structure with 10 random numbers:
////////////////////////////////////////////////////////////////////////

        NIL     <======          35      ======>         NIL
         35     <======         604      ======>        1213
        NIL     <======         810      ======>         NIL
        810     <======        1213      ======>        6152
        NIL     <======        3956      ======>         NIL
       3956     <======        6102      ======>         NIL
       6102     <======        6152      ======>        9609
        NIL     <======        6383      ======>        9221
        NIL     <======        9221      ======>         NIL
       6383     <======        9609      ======>         NIL

////////////////////////////////////////////////////////////////////////
// The following data structure is a copy of the above one:
////////////////////////////////////////////////////////////////////////

        NIL     <======          35      ======>         NIL
         35     <======         604      ======>        1213
        NIL     <======         810      ======>         NIL
        810     <======        1213      ======>        6152
        NIL     <======        3956      ======>         NIL
       3956     <======        6102      ======>         NIL
       6102     <======        6152      ======>        9609
        NIL     <======        6383      ======>        9221
        NIL     <======        9221      ======>         NIL
       6383     <======        9609      ======>         NIL

////////////////////////////////////////////////////////////////////////
// After inserting 12345, 4321, 15, 8876, 5034:
////////////////////////////////////////////////////////////////////////

        NIL     <======          15      ======>         NIL
         15     <======          35      ======>         NIL
         35     <======         604      ======>        1213
        NIL     <======         810      ======>         NIL
        810     <======        1213      ======>        6152
        NIL     <======        3956      ======>        4321
        NIL     <======        4321      ======>        5034
        NIL     <======        5034      ======>         NIL
       3956     <======        6102      ======>         NIL
       6102     <======        6152      ======>        9609
        NIL     <======        6383      ======>        9221
        NIL     <======        8876      ======>         NIL
       8876     <======        9221      ======>         NIL
       6383     <======        9609      ======>       12345
        NIL     <======       12345      ======>         NIL

////////////////////////////////////////////////////////////////////////
// Search for the newly inserted numbers
////////////////////////////////////////////////////////////////////////

Find number 12345
Find number 4321
Find number 15
Find number 8876
Find number 5034
```

```
Cannot find random number 7762


//////////////////////////////////////////////////////////////////////////
// Remove test
//////////////////////////////////////////////////////////////////////////

===After removing 12345, 4321, 15,

         NIL     <======          35     ======>         NIL
          35     <======         604     ======>        1213
         NIL     <======         810     ======>         NIL
         810     <======        1213     ======>        6152
         NIL     <======        3956     ======>        5034
         NIL     <======        5034     ======>         NIL
        3956     <======        6102     ======>         NIL
        6102     <======        6152     ======>        9609
         NIL     <======        6383     ======>        9221
         NIL     <======        8876     ======>         NIL
        8876     <======        9221     ======>         NIL
        6383     <======        9609     ======>         NIL

===After removing 8876, 5034,

         NIL     <======          35     ======>         NIL
          35     <======         604     ======>        1213
         NIL     <======         810     ======>         NIL
         810     <======        1213     ======>        6152
         NIL     <======        3956     ======>         NIL
        3956     <======        6102     ======>         NIL
        6102     <======        6152     ======>        9609
         NIL     <======        6383     ======>        9221
         NIL     <======        9221     ======>         NIL
        6383     <======        9609     ======>         NIL


//////////////////////////////////////////////////////////////////////////
// Remove a random number from the above structure.
//////////////////////////////////////////////////////////////////////////

Cannot remove 7762


//////////////////////////////////////////////////////////////////////////
// After empty(), the above data structure become:
//////////////////////////////////////////////////////////////////////////



//////////////////////////////////////////////////////////////////////////
// Measure time needed to build and access the data structure.
//////////////////////////////////////////////////////////////////////////

====== Data structure with 10000 nodes ======
Time to build the data structure: 140 milliseconds
Time to access the data structure for 10000 times: 90 milliseconds
Space required to build the data structure: 160008 bytes

====== Data structure with 30000 nodes ======
Time to build the data structure: 411 milliseconds
Time to access the data structure for 30000 times: 360 milliseconds
Space required to build the data structure: 480008 bytes

====== Data structure with 50000 nodes ======
Time to build the data structure: 731 milliseconds
Time to access the data structure for 50000 times: 621 milliseconds
Space required to build the data structure: 800008 bytes

====== Data structure with 70000 nodes ======
Time to build the data structure: 1091 milliseconds
Time to access the data structure for 70000 times: 911 milliseconds
```

```
Space required to build the data structure: 1120008 bytes

====== Data structure with 90000 nodes ======
Time to build the data structure: 1662 milliseconds
Time to access the data structure for 90000 times: 1222 milliseconds
Space required to build the data structure: 1439992 bytes

====== Data structure with 110000 nodes ======
Time to build the data structure: 1882 milliseconds
Time to access the data structure for 110000 times: 1633 milliseconds
Space required to build the data structure: 1760008 bytes

====== Data structure with 130000 nodes ======
Time to build the data structure: 2293 milliseconds
Time to access the data structure for 130000 times: 1973 milliseconds
Space required to build the data structure: 2079976 bytes

====== Data structure with 150000 nodes ======
Time to build the data structure: 2694 milliseconds
Time to access the data structure for 150000 times: 2283 milliseconds
Space required to build the data structure: 2399960 bytes

====== Data structure with 170000 nodes ======
Time to build the data structure: 3195 milliseconds
Time to access the data structure for 170000 times: 2704 milliseconds
Space required to build the data structure: 2719960 bytes

====== Data structure with 190000 nodes ======
Time to build the data structure: 3485 milliseconds
Time to access the data structure for 190000 times: 2964 milliseconds
Space required to build the data structure: 3039864 bytes

====== Data structure with 210000 nodes ======
Time to build the data structure: 3916 milliseconds
Time to access the data structure for 210000 times: 3385 milliseconds
Space required to build the data structure: 3359896 bytes

====== Data structure with 230000 nodes ======
Time to build the data structure: 4416 milliseconds
Time to access the data structure for 230000 times: 3816 milliseconds
Space required to build the data structure: 3679960 bytes

====== Data structure with 250000 nodes ======
Time to build the data structure: 4767 milliseconds
Time to access the data structure for 250000 times: 4136 milliseconds
Space required to build the data structure: 3999928 bytes

====== Data structure with 270000 nodes ======
Time to build the data structure: 5378 milliseconds
Time to access the data structure for 270000 times: 4777 milliseconds
Space required to build the data structure: 4319896 bytes

====== Data structure with 290000 nodes ======
Time to build the data structure: 5598 milliseconds
Time to access the data structure for 290000 times: 4857 milliseconds
Space required to build the data structure: 4639800 bytes

====== Data structure with 310000 nodes ======
Time to build the data structure: 6049 milliseconds
Time to access the data structure for 310000 times: 5178 milliseconds
Space required to build the data structure: 4959832 bytes
```

Test Result 3: *TestIntTmplTree.txt*

```
================================================================
 Test Template Binary Tree <int> Built With Null Node Objects
================================================================


////////////////////////////////////////////////////////////////////////
// Here is a data structure with 10 random numbers:
////////////////////////////////////////////////////////////////////////

        NIL     <======          35      ======>         NIL
         35     <======         604      ======>        1213
        NIL     <======         810      ======>         NIL
        810     <======        1213      ======>        6152
        NIL     <======        3956      ======>         NIL
       3956     <======        6102      ======>         NIL
       6102     <======        6152      ======>        9609
        NIL     <======        6383      ======>        9221
        NIL     <======        9221      ======>         NIL
       6383     <======        9609      ======>         NIL


////////////////////////////////////////////////////////////////////////
// The following data structure is a copy of the above one:
////////////////////////////////////////////////////////////////////////

        NIL     <======          35      ======>         NIL
         35     <======         604      ======>        1213
        NIL     <======         810      ======>         NIL
        810     <======        1213      ======>        6152
        NIL     <======        3956      ======>         NIL
       3956     <======        6102      ======>         NIL
       6102     <======        6152      ======>        9609
        NIL     <======        6383      ======>        9221
        NIL     <======        9221      ======>         NIL
       6383     <======        9609      ======>         NIL


////////////////////////////////////////////////////////////////////////
// After inserting 12345, 4321, 15, 8876, 5034:
////////////////////////////////////////////////////////////////////////

        NIL     <======          15      ======>         NIL
         15     <======          35      ======>         NIL
         35     <======         604      ======>        1213
        NIL     <======         810      ======>         NIL
        810     <======        1213      ======>        6152
        NIL     <======        3956      ======>        4321
        NIL     <======        4321      ======>        5034
        NIL     <======        5034      ======>         NIL
       3956     <======        6102      ======>         NIL
       6102     <======        6152      ======>        9609
        NIL     <======        6383      ======>        9221
        NIL     <======        8876      ======>         NIL
       8876     <======        9221      ======>         NIL
       6383     <======        9609      ======>       12345
        NIL     <======       12345      ======>         NIL


////////////////////////////////////////////////////////////////////////
// Search for the newly inserted numbers
////////////////////////////////////////////////////////////////////////

Find number 12345
Find number 4321
Find number 15
Find number 8876
Find number 5034
```

```
Cannot find random number 7762


//////////////////////////////////////////////////////////////////////
// Remove test
//////////////////////////////////////////////////////////////////////

===After removing 12345, 4321, 15,

         NIL     <======          35     ======>         NIL
          35     <======         604     ======>        1213
         NIL     <======         810     ======>         NIL
         810     <======        1213     ======>        6152
         NIL     <======        3956     ======>        5034
         NIL     <======        5034     ======>         NIL
        3956     <======        6102     ======>         NIL
        6102     <======        6152     ======>        9609
         NIL     <======        6383     ======>        9221
         NIL     <======        8876     ======>         NIL
        8876     <======        9221     ======>         NIL
        6383     <======        9609     ======>         NIL

===After removing 8876, 5034,

         NIL     <======          35     ======>         NIL
          35     <======         604     ======>        1213
         NIL     <======         810     ======>         NIL
         810     <======        1213     ======>        6152
         NIL     <======        3956     ======>         NIL
        3956     <======        6102     ======>         NIL
        6102     <======        6152     ======>        9609
         NIL     <======        6383     ======>        9221
         NIL     <======        9221     ======>         NIL
        6383     <======        9609     ======>         NIL

//////////////////////////////////////////////////////////////////////
// Remove a random number from the above structure.
//////////////////////////////////////////////////////////////////////

Cannot remove 7762


//////////////////////////////////////////////////////////////////////
// After empty(), the above data structure become:
//////////////////////////////////////////////////////////////////////



//////////////////////////////////////////////////////////////////////
// Measure time needed to build and access the data structure.
//////////////////////////////////////////////////////////////////////

====== Data structure with 10000 nodes ======
Time to build the data structure: 100 milliseconds
Time to access the data structure for 10000 times: 100 milliseconds
Space required to build the data structure: 160012 bytes

====== Data structure with 30000 nodes ======
Time to build the data structure: 371 milliseconds
Time to access the data structure for 30000 times: 340 milliseconds
Space required to build the data structure: 480008 bytes

====== Data structure with 50000 nodes ======
Time to build the data structure: 621 milliseconds
Time to access the data structure for 50000 times: 610 milliseconds
Space required to build the data structure: 800008 bytes

====== Data structure with 70000 nodes ======
Time to build the data structure: 941 milliseconds
Time to access the data structure for 70000 times: 921 milliseconds
```

```
Space required to build the data structure: 1120008 bytes

====== Data structure with 90000 nodes ======
Time to build the data structure: 1492 milliseconds
Time to access the data structure for 90000 times: 1202 milliseconds
Space required to build the data structure: 1439992 bytes

====== Data structure with 110000 nodes ======
Time to build the data structure: 1633 milliseconds
Time to access the data structure for 110000 times: 1582 milliseconds
Space required to build the data structure: 1760008 bytes

====== Data structure with 130000 nodes ======
Time to build the data structure: 2023 milliseconds
Time to access the data structure for 130000 times: 1932 milliseconds
Space required to build the data structure: 2079976 bytes

====== Data structure with 150000 nodes ======
Time to build the data structure: 2344 milliseconds
Time to access the data structure for 150000 times: 2293 milliseconds
Space required to build the data structure: 2399960 bytes

====== Data structure with 170000 nodes ======
Time to build the data structure: 2754 milliseconds
Time to access the data structure for 170000 times: 2694 milliseconds
Space required to build the data structure: 2719960 bytes

====== Data structure with 190000 nodes ======
Time to build the data structure: 3055 milliseconds
Time to access the data structure for 190000 times: 2974 milliseconds
Space required to build the data structure: 3039864 bytes

====== Data structure with 210000 nodes ======
Time to build the data structure: 3445 milliseconds
Time to access the data structure for 210000 times: 3335 milliseconds
Space required to build the data structure: 3359896 bytes

====== Data structure with 230000 nodes ======
Time to build the data structure: 3836 milliseconds
Time to access the data structure for 230000 times: 3725 milliseconds
Space required to build the data structure: 3679960 bytes

====== Data structure with 250000 nodes ======
Time to build the data structure: 4236 milliseconds
Time to access the data structure for 250000 times: 4096 milliseconds
Space required to build the data structure: 3999928 bytes

====== Data structure with 270000 nodes ======
Time to build the data structure: 4717 milliseconds
Time to access the data structure for 270000 times: 4546 milliseconds
Space required to build the data structure: 4319896 bytes

====== Data structure with 290000 nodes ======
Time to build the data structure: 4927 milliseconds
Time to access the data structure for 290000 times: 4757 milliseconds
Space required to build the data structure: 4639800 bytes

====== Data structure with 310000 nodes ======
Time to build the data structure: 5428 milliseconds
Time to access the data structure for 310000 times: 5227 milliseconds
Space required to build the data structure: 4959832 bytes
```

Test Result 4: *TestIntTmplPtrTree.txt*

```
============================================================
 Test Template Binary Tree <int> Built With NULL pointers
============================================================

//////////////////////////////////////////////////////////////////////
// Here is a data structure with 10 random numbers:
//////////////////////////////////////////////////////////////////////

       NIL     <======          35     ======>          NIL
        35     <======         604     ======>         1213
       NIL     <======         810     ======>          NIL
       810     <======        1213     ======>         6152
       NIL     <======        3956     ======>          NIL
      3956     <======        6102     ======>          NIL
      6102     <======        6152     ======>         9609
       NIL     <======        6383     ======>         9221
       NIL     <======        9221     ======>          NIL
      6383     <======        9609     ======>          NIL

//////////////////////////////////////////////////////////////////////
// The following data structure is a copy of the above one:
//////////////////////////////////////////////////////////////////////

       NIL     <======          35     ======>          NIL
        35     <======         604     ======>         1213
       NIL     <======         810     ======>          NIL
       810     <======        1213     ======>         6152
       NIL     <======        3956     ======>          NIL
      3956     <======        6102     ======>          NIL
      6102     <======        6152     ======>         9609
       NIL     <======        6383     ======>         9221
       NIL     <======        9221     ======>          NIL
      6383     <======        9609     ======>          NIL

//////////////////////////////////////////////////////////////////////
// After inserting 12345, 4321, 15, 8876, 5034:
//////////////////////////////////////////////////////////////////////

       NIL     <======          15     ======>          NIL
        15     <======          35     ======>          NIL
        35     <======         604     ======>         1213
       NIL     <======         810     ======>          NIL
       810     <======        1213     ======>         6152
       NIL     <======        3956     ======>         4321
       NIL     <======        4321     ======>         5034
       NIL     <======        5034     ======>          NIL
      3956     <======        6102     ======>          NIL
      6102     <======        6152     ======>         9609
       NIL     <======        6383     ======>         9221
       NIL     <======        8876     ======>          NIL
      8876     <======        9221     ======>          NIL
      6383     <======        9609     ======>        12345
       NIL     <======       12345     ======>          NIL

//////////////////////////////////////////////////////////////////////
// Search for the newly inserted numbers
//////////////////////////////////////////////////////////////////////

Find number 12345
Find number 4321
Find number 15
Find number 8876
Find number 5034
```

```
Cannot find random number 7762


////////////////////////////////////////////////////////////////////////
// Remove test
////////////////////////////////////////////////////////////////////////

===After removing 12345, 4321, 15,

         NIL     <======          35     ======>         NIL
          35     <======         604     ======>        1213
         NIL     <======         810     ======>         NIL
         810     <======        1213     ======>        6152
         NIL     <======        3956     ======>        5034
         NIL     <======        5034     ======>         NIL
        3956     <======        6102     ======>         NIL
        6102     <======        6152     ======>        9609
         NIL     <======        6383     ======>        9221
         NIL     <======        8876     ======>         NIL
        8876     <======        9221     ======>         NIL
        6383     <======        9609     ======>         NIL

===After removing 8876, 5034,

         NIL     <======          35     ======>         NIL
          35     <======         604     ======>        1213
         NIL     <======         810     ======>         NIL
         810     <======        1213     ======>        6152
         NIL     <======        3956     ======>         NIL
        3956     <======        6102     ======>         NIL
        6102     <======        6152     ======>        9609
         NIL     <======        6383     ======>        9221
         NIL     <======        9221     ======>         NIL
        6383     <======        9609     ======>         NIL


////////////////////////////////////////////////////////////////////////
// Remove a random number from the above structure.
////////////////////////////////////////////////////////////////////////

Cannot remove 7762


////////////////////////////////////////////////////////////////////////
// After empty(), the above data structure become:
////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////
// Measure time needed to build and access the data structure.
////////////////////////////////////////////////////////////////////////

====== Data structure with 10000 nodes ======
Time to build the data structure: 130 milliseconds
Time to access the data structure for 10000 times: 100 milliseconds
Space required to build the data structure: 160008 bytes

====== Data structure with 30000 nodes ======
Time to build the data structure: 431 milliseconds
Time to access the data structure for 30000 times: 330 milliseconds
Space required to build the data structure: 480008 bytes

====== Data structure with 50000 nodes ======
Time to build the data structure: 741 milliseconds
Time to access the data structure for 50000 times: 611 milliseconds
Space required to build the data structure: 800008 bytes

====== Data structure with 70000 nodes ======
Time to build the data structure: 1111 milliseconds
Time to access the data structure for 70000 times: 921 milliseconds
```

```
Space required to build the data structure: 1120008 bytes

====== Data structure with 90000 nodes ======
Time to build the data structure: 1632 milliseconds
Time to access the data structure for 90000 times: 1192 milliseconds
Space required to build the data structure: 1439992 bytes

====== Data structure with 110000 nodes ======
Time to build the data structure: 1893 milliseconds
Time to access the data structure for 110000 times: 1612 milliseconds
Space required to build the data structure: 1760008 bytes

====== Data structure with 130000 nodes ======
Time to build the data structure: 2303 milliseconds
Time to access the data structure for 130000 times: 1923 milliseconds
Space required to build the data structure: 2079976 bytes

====== Data structure with 150000 nodes ======
Time to build the data structure: 2724 milliseconds
Time to access the data structure for 150000 times: 2263 milliseconds
Space required to build the data structure: 2399960 bytes

====== Data structure with 170000 nodes ======
Time to build the data structure: 3215 milliseconds
Time to access the data structure for 170000 times: 2723 milliseconds
Space required to build the data structure: 2719960 bytes

====== Data structure with 190000 nodes ======
Time to build the data structure: 3485 milliseconds
Time to access the data structure for 190000 times: 2975 milliseconds
Space required to build the data structure: 3039864 bytes

====== Data structure with 210000 nodes ======
Time to build the data structure: 3895 milliseconds
Time to access the data structure for 210000 times: 3325 milliseconds
Space required to build the data structure: 3359896 bytes

====== Data structure with 230000 nodes ======
Time to build the data structure: 4397 milliseconds
Time to access the data structure for 230000 times: 3765 milliseconds
Space required to build the data structure: 3679960 bytes

====== Data structure with 250000 nodes ======
Time to build the data structure: 4756 milliseconds
Time to access the data structure for 250000 times: 4096 milliseconds
Space required to build the data structure: 3999928 bytes

====== Data structure with 270000 nodes ======
Time to build the data structure: 5347 milliseconds
Time to access the data structure for 270000 times: 4577 milliseconds
Space required to build the data structure: 4319896 bytes

====== Data structure with 290000 nodes ======
Time to build the data structure: 5638 milliseconds
Time to access the data structure for 290000 times: 4857 milliseconds
Space required to build the data structure: 4639800 bytes

====== Data structure with 310000 nodes ======
Time to build the data structure: 6099 milliseconds
Time to access the data structure for 310000 times: 5268 milliseconds
Space required to build the data structure: 4959832 bytes
```

Test Result 5: *TestAVLTree.txt*

```
================================================
 Test AVL Binary Tree Built With Null Node Object
================================================


////////////////////////////////////////////////////////////////////////
// Here is a height balance tree with numbers 1 to 100 (right rotation):
////////////////////////////////////////////////////////////////////////

        NIL      <======          1      ======>          NIL
          1      <======          2      ======>            3
        NIL      <======          3      ======>          NIL
          2      <======          4      ======>            6
        NIL      <======          5      ======>          NIL
          5      <======          6      ======>            7
        NIL      <======          7      ======>          NIL
          4      <======          8      ======>           12
        NIL      <======          9      ======>          NIL
          9      <======         10      ======>           11
        NIL      <======         11      ======>          NIL
         10      <======         12      ======>           14
        NIL      <======         13      ======>          NIL
         13      <======         14      ======>           15
        NIL      <======         15      ======>          NIL
          8      <======         16      ======>           24
        NIL      <======         17      ======>          NIL
         17      <======         18      ======>           19
        NIL      <======         19      ======>          NIL
         18      <======         20      ======>           22
        NIL      <======         21      ======>          NIL
         21      <======         22      ======>           23
        NIL      <======         23      ======>          NIL
         20      <======         24      ======>           28
        NIL      <======         25      ======>          NIL
         25      <======         26      ======>           27
        NIL      <======         27      ======>          NIL
         26      <======         28      ======>           30
        NIL      <======         29      ======>          NIL
         29      <======         30      ======>           31
        NIL      <======         31      ======>          NIL
         16      <======         32      ======>           48
        NIL      <======         33      ======>          NIL
         33      <======         34      ======>           35
        NIL      <======         35      ======>          NIL
         34      <======         36      ======>           38
        NIL      <======         37      ======>          NIL
         37      <======         38      ======>           39
        NIL      <======         39      ======>          NIL
         36      <======         40      ======>           44
        NIL      <======         41      ======>          NIL
         41      <======         42      ======>           43
        NIL      <======         43      ======>          NIL
         42      <======         44      ======>           46
        NIL      <======         45      ======>          NIL
         45      <======         46      ======>           47
        NIL      <======         47      ======>          NIL
         40      <======         48      ======>           56
        NIL      <======         49      ======>          NIL
         49      <======         50      ======>           51
        NIL      <======         51      ======>          NIL
         50      <======         52      ======>           54
        NIL      <======         53      ======>          NIL
         53      <======         54      ======>           55
        NIL      <======         55      ======>          NIL
         52      <======         56      ======>           60
        NIL      <======         57      ======>          NIL
         57      <======         58      ======>           59
```

```
NIL     <======          59      ======>          NIL
 58     <======          60      ======>           62
NIL     <======          61      ======>          NIL
 61     <======          62      ======>           63
NIL     <======          63      ======>          NIL
 32     <======          64      ======>           80
NIL     <======          65      ======>          NIL
 65     <======          66      ======>           67
NIL     <======          67      ======>          NIL
 66     <======          68      ======>           70
NIL     <======          69      ======>          NIL
 69     <======          70      ======>           71
NIL     <======          71      ======>          NIL
 68     <======          72      ======>           76
NIL     <======          73      ======>          NIL
 73     <======          74      ======>           75
NIL     <======          75      ======>          NIL
 74     <======          76      ======>           78
NIL     <======          77      ======>          NIL
 77     <======          78      ======>           79
NIL     <======          79      ======>          NIL
 72     <======          80      ======>           88
NIL     <======          81      ======>          NIL
 81     <======          82      ======>           83
NIL     <======          83      ======>          NIL
 82     <======          84      ======>           86
NIL     <======          85      ======>          NIL
 85     <======          86      ======>           87
NIL     <======          87      ======>          NIL
 84     <======          88      ======>           96
NIL     <======          89      ======>          NIL
 89     <======          90      ======>           91
NIL     <======          91      ======>          NIL
 90     <======          92      ======>           94
NIL     <======          93      ======>          NIL
 93     <======          94      ======>           95
NIL     <======          95      ======>          NIL
 92     <======          96      ======>           98
NIL     <======          97      ======>          NIL
 97     <======          98      ======>           99
NIL     <======          99      ======>          100
NIL     <======         100      ======>          NIL
```

//
After removing 4, 13, 10, 35, 52, 9, 2, 83, 56, 21, 62, 91, 32, 17, 94, 83, 56, 37, 66, 51, 80,
//

```
NIL     <======           1      ======>          NIL
  1     <======           3      ======>          NIL
  3     <======           5      ======>            6
NIL     <======           6      ======>            7
NIL     <======           7      ======>          NIL
  5     <======           8      ======>           12
NIL     <======          11      ======>          NIL
 11     <======          12      ======>           14
NIL     <======          14      ======>           15
NIL     <======          15      ======>          NIL
  8     <======          16      ======>           24
NIL     <======          18      ======>           19
NIL     <======          19      ======>          NIL
 18     <======          20      ======>           22
NIL     <======          22      ======>           23
NIL     <======          23      ======>          NIL
 20     <======          24      ======>           28
NIL     <======          25      ======>          NIL
 25     <======          26      ======>           27
NIL     <======          27      ======>          NIL
 26     <======          28      ======>           30
NIL     <======          29      ======>          NIL
```

```
 29     <======          30     ======>          31
NIL     <======          31     ======>         NIL
 16     <======          33     ======>          48
NIL     <======          34     ======>         NIL
 34     <======          36     ======>          38
NIL     <======          38     ======>          39
NIL     <======          39     ======>         NIL
 36     <======          40     ======>          44
NIL     <======          41     ======>         NIL
 41     <======          42     ======>          43
NIL     <======          43     ======>         NIL
 42     <======          44     ======>          46
NIL     <======          45     ======>         NIL
 45     <======          46     ======>          47
NIL     <======          47     ======>         NIL
 40     <======          48     ======>          57
NIL     <======          49     ======>         NIL
 49     <======          50     ======>         NIL
 50     <======          53     ======>          54
NIL     <======          54     ======>          55
NIL     <======          55     ======>         NIL
 53     <======          57     ======>          60
NIL     <======          58     ======>          59
NIL     <======          59     ======>         NIL
 58     <======          60     ======>          63
NIL     <======          61     ======>         NIL
 61     <======          63     ======>         NIL
 33     <======          64     ======>          81
NIL     <======          65     ======>         NIL
 65     <======          67     ======>         NIL
 67     <======          68     ======>          70
NIL     <======          69     ======>         NIL
 69     <======          70     ======>          71
NIL     <======          71     ======>         NIL
 68     <======          72     ======>          76
NIL     <======          73     ======>         NIL
 73     <======          74     ======>          75
NIL     <======          75     ======>         NIL
 74     <======          76     ======>          78
NIL     <======          77     ======>         NIL
 77     <======          78     ======>          79
NIL     <======          79     ======>         NIL
 72     <======          81     ======>          88
NIL     <======          82     ======>         NIL
 82     <======          84     ======>          86
NIL     <======          85     ======>         NIL
 85     <======          86     ======>          87
NIL     <======          87     ======>         NIL
 84     <======          88     ======>          96
NIL     <======          89     ======>         NIL
 89     <======          90     ======>         NIL
 90     <======          92     ======>          95
NIL     <======          93     ======>         NIL
 93     <======          95     ======>         NIL
 92     <======          96     ======>          98
NIL     <======          97     ======>         NIL
 97     <======          98     ======>          99
NIL     <======          99     ======>         100
NIL     <======         100     ======>         NIL
```

```
//////////////////////////////////////////////////////////////////////
After removing 77, 66, 47, 72, 49, 70, 87, 48, 61, 86, 7, 32, 21, 62, 31, 92, 45, 94, 15,
96,
//////////////////////////////////////////////////////////////////////
```

```
NIL     <======           1     ======>         NIL
  1     <======           3     ======>         NIL
  3     <======           5     ======>           6
NIL     <======           6     ======>         NIL
  5     <======           8     ======>          12
```

NIL	<======	11	======>	NIL
11	<======	12	======>	14
NIL	<======	14	======>	NIL
8	<======	16	======>	24
NIL	<======	18	======>	19
NIL	<======	19	======>	NIL
18	<======	20	======>	22
NIL	<======	22	======>	23
NIL	<======	23	======>	NIL
20	<======	24	======>	28
NIL	<======	25	======>	NIL
25	<======	26	======>	27
NIL	<======	27	======>	NIL
26	<======	28	======>	30
NIL	<======	29	======>	NIL
29	<======	30	======>	NIL
16	<======	33	======>	50
NIL	<======	34	======>	NIL
34	<======	36	======>	38
NIL	<======	38	======>	39
NIL	<======	39	======>	NIL
36	<======	40	======>	44
NIL	<======	41	======>	NIL
41	<======	42	======>	43
NIL	<======	43	======>	NIL
42	<======	44	======>	46
NIL	<======	46	======>	NIL
40	<======	50	======>	57
NIL	<======	53	======>	NIL
53	<======	54	======>	55
NIL	<======	55	======>	NIL
54	<======	57	======>	60
NIL	<======	58	======>	59
NIL	<======	59	======>	NIL
58	<======	60	======>	63
NIL	<======	63	======>	NIL
33	<======	64	======>	81
NIL	<======	65	======>	NIL
65	<======	67	======>	NIL
67	<======	68	======>	71
NIL	<======	69	======>	NIL
69	<======	71	======>	NIL
68	<======	73	======>	76
NIL	<======	74	======>	75
NIL	<======	75	======>	NIL
74	<======	76	======>	78
NIL	<======	78	======>	79
NIL	<======	79	======>	NIL
73	<======	81	======>	93
NIL	<======	82	======>	NIL
82	<======	84	======>	85
NIL	<======	85	======>	NIL
84	<======	88	======>	90
NIL	<======	89	======>	NIL
89	<======	90	======>	NIL
88	<======	93	======>	97
NIL	<======	95	======>	NIL
95	<======	97	======>	99
NIL	<======	98	======>	NIL
98	<======	99	======>	100
NIL	<======	100	======>	NIL

//
After removing 49, 14, 51, 4, 85, 78, 43, 96, 85, 38, 71, 0, 33, 58, 67, 56, 65, 62, 55, 20,
//

NIL	<======	1	======>	NIL
1	<======	3	======>	NIL
3	<======	5	======>	6

```
 NIL     <======          6     ======>     NIL
   5     <======          8     ======>      12
 NIL     <======         11     ======>     NIL
  11     <======         12     ======>     NIL
   8     <======         16     ======>      24
 NIL     <======         18     ======>      19
 NIL     <======         19     ======>     NIL
  18     <======         22     ======>      23
 NIL     <======         23     ======>     NIL
  22     <======         24     ======>      28
 NIL     <======         25     ======>     NIL
  25     <======         26     ======>      27
 NIL     <======         27     ======>     NIL
  26     <======         28     ======>      30
 NIL     <======         29     ======>     NIL
  29     <======         30     ======>     NIL
  16     <======         34     ======>      50
 NIL     <======         36     ======>      39
 NIL     <======         39     ======>     NIL
  36     <======         40     ======>      44
 NIL     <======         41     ======>     NIL
  41     <======         42     ======>     NIL
  42     <======         44     ======>      46
 NIL     <======         46     ======>     NIL
  40     <======         50     ======>      57
 NIL     <======         53     ======>     NIL
  53     <======         54     ======>     NIL
  54     <======         57     ======>      60
 NIL     <======         59     ======>     NIL
  59     <======         60     ======>      63
 NIL     <======         63     ======>     NIL
  34     <======         64     ======>      81
 NIL     <======         68     ======>      69
 NIL     <======         69     ======>     NIL
  68     <======         73     ======>      76
 NIL     <======         74     ======>      75
 NIL     <======         75     ======>     NIL
  74     <======         76     ======>      79
 NIL     <======         79     ======>     NIL
  73     <======         81     ======>      93
 NIL     <======         82     ======>     NIL
  82     <======         84     ======>     NIL
  84     <======         88     ======>      90
 NIL     <======         89     ======>     NIL
  89     <======         90     ======>     NIL
  88     <======         93     ======>      97
 NIL     <======         95     ======>     NIL
  95     <======         97     ======>      99
 NIL     <======         98     ======>     NIL
  98     <======         99     ======>     100
 NIL     <======        100     ======>     NIL
```

```
////////////////////////////////////////////////////////////////////
After removing 45, 2, 83, 64, 29, 18, 99, 40, 57, 50, 63, 36, 45, 54, 3, 0, 89, 62, 59,
24,
////////////////////////////////////////////////////////////////////
```

```
 NIL     <======          1     ======>     NIL
   1     <======          5     ======>       6
 NIL     <======          6     ======>     NIL
   5     <======          8     ======>      12
 NIL     <======         11     ======>     NIL
  11     <======         12     ======>     NIL
   8     <======         16     ======>      22
 NIL     <======         19     ======>     NIL
  19     <======         22     ======>      23
 NIL     <======         23     ======>     NIL
  16     <======         25     ======>      34
 NIL     <======         26     ======>      27
 NIL     <======         27     ======>     NIL
```

```
 26     <======          28     ======>          30
NIL     <======          30     ======>         NIL
 28     <======          34     ======>          44
NIL     <======          39     ======>         NIL
 39     <======          41     ======>          42
NIL     <======          42     ======>         NIL
 41     <======          44     ======>          53
NIL     <======          46     ======>         NIL
 46     <======          53     ======>          60
NIL     <======          60     ======>         NIL
 25     <======          68     ======>          81
NIL     <======          69     ======>         NIL
 69     <======          73     ======>         NIL
 73     <======          74     ======>          76
NIL     <======          75     ======>         NIL
 75     <======          76     ======>          79
NIL     <======          79     ======>         NIL
 74     <======          81     ======>          93
NIL     <======          82     ======>         NIL
 82     <======          84     ======>         NIL
 84     <======          88     ======>          90
NIL     <======          90     ======>         NIL
 88     <======          93     ======>          97
NIL     <======          95     ======>         NIL
 95     <======          97     ======>         100
NIL     <======          98     ======>         NIL
 98     <======         100     ======>         NIL
```

```
////////////////////////////////////////////////////////////////////
After removing 97, 70, 59, 76, 1, 74, 47, 8, 1, 54, 7, 36, 85, 98, 23, 52, 97, 70, 11,
44,
////////////////////////////////////////////////////////////////////
```

```
NIL     <======           5     ======>         NIL
  5     <======           6     ======>          12
NIL     <======          12     ======>         NIL
  6     <======          16     ======>          22
NIL     <======          19     ======>         NIL
 19     <======          22     ======>         NIL
 16     <======          25     ======>          34
NIL     <======          26     ======>          27
NIL     <======          27     ======>         NIL
 26     <======          28     ======>          30
NIL     <======          30     ======>         NIL
 28     <======          34     ======>          46
NIL     <======          39     ======>         NIL
 39     <======          41     ======>          42
NIL     <======          42     ======>         NIL
 41     <======          46     ======>          53
NIL     <======          53     ======>          60
NIL     <======          60     ======>         NIL
 25     <======          68     ======>          81
NIL     <======          69     ======>         NIL
 69     <======          73     ======>         NIL
 73     <======          75     ======>          79
NIL     <======          79     ======>         NIL
 75     <======          81     ======>          93
NIL     <======          82     ======>         NIL
 82     <======          84     ======>         NIL
 84     <======          88     ======>          90
NIL     <======          90     ======>         NIL
 88     <======          93     ======>         100
NIL     <======          95     ======>         NIL
 95     <======         100     ======>         NIL
```

```
////////////////////////////////////////////////////////////////////
After removing 33, 98, 35, 36, 33, 30, 95, 48, 77, 46, 55, 40, 61, 90, 23, 44, 69, 30,
19, 12,
////////////////////////////////////////////////////////////////////
```

```
NIL     <======          5     ======>        NIL
  5     <======          6     ======>        NIL
  6     <======         16     ======>         22
NIL     <======         22     ======>        NIL
 16     <======         25     ======>         34
NIL     <======         26     ======>        NIL
 26     <======         27     ======>         28
NIL     <======         28     ======>        NIL
 27     <======         34     ======>         53
NIL     <======         39     ======>        NIL
 39     <======         41     ======>         42
NIL     <======         42     ======>        NIL
 41     <======         53     ======>         60
NIL     <======         60     ======>        NIL
 25     <======         68     ======>         81
NIL     <======         73     ======>        NIL
 73     <======         75     ======>         79
NIL     <======         79     ======>        NIL
 75     <======         81     ======>         88
NIL     <======         82     ======>        NIL
 82     <======         84     ======>        NIL
 84     <======         88     ======>         93
NIL     <======         93     ======>        100
NIL     <======        100     ======>        NIL
```

```
////////////////////////////////////////////////////////////////////////
After removing 89, 6, 7, 84, 17, 26, 59, 0, 77, 46, 71, 76, 21, 74, 51, 52, 93, 62, 67,
48,
////////////////////////////////////////////////////////////////////////
```

```
NIL     <======          5     ======>        NIL
  5     <======         16     ======>         22
NIL     <======         22     ======>        NIL
 16     <======         25     ======>         27
NIL     <======         27     ======>         28
NIL     <======         28     ======>        NIL
 25     <======         34     ======>         53
NIL     <======         39     ======>        NIL
 39     <======         41     ======>         42
NIL     <======         42     ======>        NIL
 41     <======         53     ======>         60
NIL     <======         60     ======>        NIL
 34     <======         68     ======>         81
NIL     <======         73     ======>        NIL
 73     <======         75     ======>         79
NIL     <======         79     ======>        NIL
 75     <======         81     ======>         88
NIL     <======         82     ======>        NIL
 82     <======         88     ======>        100
NIL     <======        100     ======>        NIL
```

```
////////////////////////////////////////////////////////////////////////
After removing 17, 82, 35, 92, 53, 70, 19, 56, 13, 26, 39, 40, 1, 62, 7, 12, 93, 86, 95,
44,
////////////////////////////////////////////////////////////////////////
```

```
NIL     <======          5     ======>        NIL
  5     <======         16     ======>         22
NIL     <======         22     ======>        NIL
 16     <======         25     ======>         27
NIL     <======         27     ======>         28
NIL     <======         28     ======>        NIL
 25     <======         34     ======>         42
NIL     <======         41     ======>        NIL
 41     <======         42     ======>         60
NIL     <======         60     ======>        NIL
 34     <======         68     ======>         81
NIL     <======         73     ======>        NIL
```

```
 73      <======          75      ======>          79
NIL      <======          79      ======>         NIL
 75      <======          81      ======>          88
NIL      <======          88      ======>         100
NIL      <======         100      ======>         NIL
```

```
////////////////////////////////////////////////////////////////////
After removing 33, 94, 23, 28, 53, 14, 43, 28, 45, 58, 39, 56, 5, 78, 47, 64, 61, 2, 31,
88,
////////////////////////////////////////////////////////////////////
```

```
NIL      <======          16      ======>          22
NIL      <======          22      ======>         NIL
 16      <======          25      ======>          27
NIL      <======          27      ======>         NIL
 25      <======          34      ======>          42
NIL      <======          41      ======>         NIL
 41      <======          42      ======>          60
NIL      <======          60      ======>         NIL
 34      <======          68      ======>          81
NIL      <======          73      ======>         NIL
 73      <======          75      ======>          79
NIL      <======          79      ======>         NIL
 75      <======          81      ======>         100
NIL      <======         100      ======>         NIL
```

```
////////////////////////////////////////////////////////////////////
// Here is a height balance tree with numbers 30 to 1 (left rotation):
////////////////////////////////////////////////////////////////////
```

```
NIL      <======           1      ======>         NIL
  1      <======           2      ======>           3
NIL      <======           3      ======>         NIL
  2      <======           4      ======>           5
NIL      <======           5      ======>         NIL
  4      <======           6      ======>          10
NIL      <======           7      ======>         NIL
  7      <======           8      ======>           9
NIL      <======           9      ======>         NIL
  8      <======          10      ======>          12
NIL      <======          11      ======>         NIL
 11      <======          12      ======>          13
NIL      <======          13      ======>         NIL
  6      <======          14      ======>          22
NIL      <======          15      ======>         NIL
 15      <======          16      ======>          17
NIL      <======          17      ======>         NIL
 16      <======          18      ======>          20
NIL      <======          19      ======>         NIL
 19      <======          20      ======>          21
NIL      <======          21      ======>         NIL
 18      <======          22      ======>          25
NIL      <======          23      ======>         NIL
 23      <======          24      ======>         NIL
 24      <======          25      ======>          27
NIL      <======          26      ======>         NIL
 26      <======          27      ======>          28
NIL      <======          28      ======>         NIL
 14      <======          29      ======>          73
NIL      <======          30      ======>         NIL
 30      <======          41      ======>          42
NIL      <======          42      ======>          60
NIL      <======          60      ======>         NIL
 41      <======          73      ======>          81
NIL      <======          75      ======>          79
NIL      <======          79      ======>         NIL
 75      <======          81      ======>         100
NIL      <======         100      ======>         NIL
```

```
////////////////////////////////////////////////////////////////////////
// Here is a height balance tree with double rotation
////////////////////////////////////////////////////////////////////////

        NIL     <======          5     ======>         NIL
          5     <======          9     ======>          10
        NIL     <======         10     ======>         NIL
          9     <======         30     ======>          51
        NIL     <======         50     ======>         NIL
         50     <======         51     ======>          53
        NIL     <======         52     ======>         NIL
         52     <======         53     ======>          55
        NIL     <======         55     ======>         NIL

////////////////////////////////////////////////////////////////////////
// Here is a data structure with 10 random numbers:
////////////////////////////////////////////////////////////////////////

        NIL     <======         35     ======>         NIL
         35     <======        604     ======>         NIL
        604     <======        810     ======>        6152
        NIL     <======       1213     ======>         NIL
       1213     <======       3956     ======>        6102
        NIL     <======       6102     ======>         NIL
       3956     <======       6152     ======>        9221
        NIL     <======       6383     ======>         NIL
       6383     <======       9221     ======>        9609
        NIL     <======       9609     ======>         NIL

////////////////////////////////////////////////////////////////////////
// The following data structure is a copy of the above one:
////////////////////////////////////////////////////////////////////////

        NIL     <======         35     ======>         NIL
         35     <======        604     ======>         NIL
        604     <======        810     ======>        6152
        NIL     <======       1213     ======>         NIL
       1213     <======       3956     ======>        6102
        NIL     <======       6102     ======>         NIL
       3956     <======       6152     ======>        9221
        NIL     <======       6383     ======>         NIL
       6383     <======       9221     ======>        9609
        NIL     <======       9609     ======>         NIL

////////////////////////////////////////////////////////////////////////
// After inserting 12345, 4321, 15, 8876, 5034:
////////////////////////////////////////////////////////////////////////

        NIL     <======         15     ======>         NIL
         15     <======         35     ======>         604
        NIL     <======        604     ======>         NIL
         35     <======        810     ======>        3956
        NIL     <======       1213     ======>         NIL
       1213     <======       3956     ======>        5034
        NIL     <======       4321     ======>         NIL
       4321     <======       5034     ======>        6102
        NIL     <======       6102     ======>         NIL
        810     <======       6152     ======>        9221
        NIL     <======       6383     ======>        8876
        NIL     <======       8876     ======>         NIL
       6383     <======       9221     ======>        9609
        NIL     <======       9609     ======>       12345
        NIL     <======      12345     ======>         NIL

////////////////////////////////////////////////////////////////////////
// Search for the newly inserted numbers
```

```
//////////////////////////////////////////////////////////////////////

Find number 12345
Find number 4321
Find number 15
Find number 8876
Find number 5034
Cannot find random number 7762


//////////////////////////////////////////////////////////////////////
// Remove test
//////////////////////////////////////////////////////////////////////

===After removing 12345, 4321, 15,

         NIL     <======          35     ======>         604
         NIL     <======         604     ======>         NIL
          35     <======         810     ======>        3956
         NIL     <======        1213     ======>         NIL
        1213     <======        3956     ======>        5034
         NIL     <======        5034     ======>        6102
         NIL     <======        6102     ======>         NIL
         810     <======        6152     ======>        9221
         NIL     <======        6383     ======>        8876
         NIL     <======        8876     ======>         NIL
        6383     <======        9221     ======>        9609
         NIL     <======        9609     ======>         NIL

===After removing 8876, 5034,

         NIL     <======          35     ======>         604
         NIL     <======         604     ======>         NIL
          35     <======         810     ======>        1213
         NIL     <======        1213     ======>         NIL
         810     <======        3956     ======>        6152
         NIL     <======        6102     ======>         NIL
        6102     <======        6152     ======>        9221
         NIL     <======        6383     ======>         NIL
        6383     <======        9221     ======>        9609
         NIL     <======        9609     ======>         NIL


//////////////////////////////////////////////////////////////////////
// Remove a random number from the above structure.
//////////////////////////////////////////////////////////////////////

Cannot remove 7762


//////////////////////////////////////////////////////////////////////
// After empty(), the above data structure become:
//////////////////////////////////////////////////////////////////////



//////////////////////////////////////////////////////////////////////
// Measure time needed to build and access the data structure.
//////////////////////////////////////////////////////////////////////

====== Data structure with 10000 nodes ======
Time to build the data structure: 521 milliseconds
Time to access the data structure for 10000 times: 80 milliseconds
Space required to build the data structure: 200012 bytes

====== Data structure with 30000 nodes ======
Time to build the data structure: 1742 milliseconds
Time to access the data structure for 30000 times: 301 milliseconds
Space required to build the data structure: 600008 bytes

====== Data structure with 50000 nodes ======
```

```
Time to build the data structure: 3315 milliseconds
Time to access the data structure for 50000 times: 550 milliseconds
Space required to build the data structure: 1000008 bytes

====== Data structure with 70000 nodes ======
Time to build the data structure: 4436 milliseconds
Time to access the data structure for 70000 times: 801 milliseconds
Space required to build the data structure: 1400008 bytes

====== Data structure with 90000 nodes ======
Time to build the data structure: 5848 milliseconds
Time to access the data structure for 90000 times: 1082 milliseconds
Space required to build the data structure: 1799988 bytes

====== Data structure with 110000 nodes ======
Time to build the data structure: 7340 milliseconds
Time to access the data structure for 110000 times: 1352 milliseconds
Space required to build the data structure: 2200008 bytes

====== Data structure with 130000 nodes ======
Time to build the data structure: 8833 milliseconds
Time to access the data structure for 130000 times: 1652 milliseconds
Space required to build the data structure: 2599968 bytes

====== Data structure with 150000 nodes ======
Time to build the data structure: 10365 milliseconds
Time to access the data structure for 150000 times: 1943 milliseconds
Space required to build the data structure: 2999948 bytes

====== Data structure with 170000 nodes ======
Time to build the data structure: 11837 milliseconds
Time to access the data structure for 170000 times: 2243 milliseconds
Space required to build the data structure: 3399948 bytes

====== Data structure with 190000 nodes ======
Time to build the data structure: 13379 milliseconds
Time to access the data structure for 190000 times: 2564 milliseconds
Space required to build the data structure: 3799828 bytes

====== Data structure with 210000 nodes ======
Time to build the data structure: 14962 milliseconds
Time to access the data structure for 210000 times: 2894 milliseconds
Space required to build the data structure: 4199868 bytes

====== Data structure with 230000 nodes ======
Time to build the data structure: 16493 milliseconds
Time to access the data structure for 230000 times: 3175 milliseconds
Space required to build the data structure: 4599948 bytes

====== Data structure with 250000 nodes ======
Time to build the data structure: 18156 milliseconds
Time to access the data structure for 250000 times: 3485 milliseconds
Space required to build the data structure: 4999908 bytes

====== Data structure with 270000 nodes ======
Time to build the data structure: 19708 milliseconds
Time to access the data structure for 270000 times: 3796 milliseconds
Space required to build the data structure: 5399868 bytes

====== Data structure with 290000 nodes ======
Time to build the data structure: 21421 milliseconds
Time to access the data structure for 290000 times: 4246 milliseconds
Space required to build the data structure: 5799748 bytes

====== Data structure with 310000 nodes ======
Time to build the data structure: 22873 milliseconds
Time to access the data structure for 310000 times: 4416 milliseconds
Space required to build the data structure: 6199788 bytes
```

Test Result 6: *TestPtrAVLTree.txt*

```
=============================================
 Test AVL Binary Tree Built With NULL pointers
=============================================


////////////////////////////////////////////////////////////////////////
// Here is a height balance tree with numbers 1 to 100 (right rotation):
////////////////////////////////////////////////////////////////////////

       NIL     <======          1     ======>          NIL
         1     <======          2     ======>            3
       NIL     <======          3     ======>          NIL
         2     <======          4     ======>            6
       NIL     <======          5     ======>          NIL
         5     <======          6     ======>            7
       NIL     <======          7     ======>          NIL
         4     <======          8     ======>           12
       NIL     <======          9     ======>          NIL
         9     <======         10     ======>           11
       NIL     <======         11     ======>          NIL
        10     <======         12     ======>           14
       NIL     <======         13     ======>          NIL
        13     <======         14     ======>           15
       NIL     <======         15     ======>          NIL
         8     <======         16     ======>           24
       NIL     <======         17     ======>          NIL
        17     <======         18     ======>           19
       NIL     <======         19     ======>          NIL
        18     <======         20     ======>           22
       NIL     <======         21     ======>          NIL
        21     <======         22     ======>           23
       NIL     <======         23     ======>          NIL
        20     <======         24     ======>           28
       NIL     <======         25     ======>          NIL
        25     <======         26     ======>           27
       NIL     <======         27     ======>          NIL
        26     <======         28     ======>           30
       NIL     <======         29     ======>          NIL
        29     <======         30     ======>           31
       NIL     <======         31     ======>          NIL
        16     <======         32     ======>           48
       NIL     <======         33     ======>          NIL
        33     <======         34     ======>           35
       NIL     <======         35     ======>          NIL
        34     <======         36     ======>           38
       NIL     <======         37     ======>          NIL
        37     <======         38     ======>           39
       NIL     <======         39     ======>          NIL
        36     <======         40     ======>           44
       NIL     <======         41     ======>          NIL
        41     <======         42     ======>           43
       NIL     <======         43     ======>          NIL
        42     <======         44     ======>           46
       NIL     <======         45     ======>          NIL
        45     <======         46     ======>           47
       NIL     <======         47     ======>          NIL
        40     <======         48     ======>           56
       NIL     <======         49     ======>          NIL
        49     <======         50     ======>           51
       NIL     <======         51     ======>          NIL
        50     <======         52     ======>           54
       NIL     <======         53     ======>          NIL
        53     <======         54     ======>           55
       NIL     <======         55     ======>          NIL
        52     <======         56     ======>           60
       NIL     <======         57     ======>          NIL
        57     <======         58     ======>           59
```

```
NIL     <======          59      ======>          NIL
 58     <======          60      ======>           62
NIL     <======          61      ======>          NIL
 61     <======          62      ======>           63
NIL     <======          63      ======>          NIL
 32     <======          64      ======>           80
NIL     <======          65      ======>          NIL
 65     <======          66      ======>           67
NIL     <======          67      ======>          NIL
 66     <======          68      ======>           70
NIL     <======          69      ======>          NIL
 69     <======          70      ======>           71
NIL     <======          71      ======>          NIL
 68     <======          72      ======>           76
NIL     <======          73      ======>          NIL
 73     <======          74      ======>           75
NIL     <======          75      ======>          NIL
 74     <======          76      ======>           78
NIL     <======          77      ======>          NIL
 77     <======          78      ======>           79
NIL     <======          79      ======>          NIL
 72     <======          80      ======>           88
NIL     <======          81      ======>          NIL
 81     <======          82      ======>           83
NIL     <======          83      ======>          NIL
 82     <======          84      ======>           86
NIL     <======          85      ======>          NIL
 85     <======          86      ======>           87
NIL     <======          87      ======>          NIL
 84     <======          88      ======>           96
NIL     <======          89      ======>          NIL
 89     <======          90      ======>           91
NIL     <======          91      ======>          NIL
 90     <======          92      ======>           94
NIL     <======          93      ======>          NIL
 93     <======          94      ======>           95
NIL     <======          95      ======>          NIL
 92     <======          96      ======>           98
NIL     <======          97      ======>          NIL
 97     <======          98      ======>           99
NIL     <======          99      ======>          100
NIL     <======         100      ======>          NIL
```

//

After removing 4, 13, 10, 35, 52, 9, 2, 83, 56, 21, 62, 91, 32, 17, 94, 83, 56, 37, 66, 51, 80,

//

```
NIL     <======           1      ======>          NIL
  1     <======           3      ======>          NIL
  3     <======           5      ======>            6
NIL     <======           6      ======>            7
NIL     <======           7      ======>          NIL
  5     <======           8      ======>           12
NIL     <======          11      ======>          NIL
 11     <======          12      ======>           14
NIL     <======          14      ======>           15
NIL     <======          15      ======>          NIL
  8     <======          16      ======>           24
NIL     <======          18      ======>           19
NIL     <======          19      ======>          NIL
 18     <======          20      ======>           22
NIL     <======          22      ======>           23
NIL     <======          23      ======>          NIL
 20     <======          24      ======>           28
NIL     <======          25      ======>          NIL
 25     <======          26      ======>           27
NIL     <======          27      ======>          NIL
 26     <======          28      ======>           30
NIL     <======          29      ======>          NIL
```

```
 29     <======          30     ======>          31
NIL     <======          31     ======>         NIL
 16     <======          33     ======>          48
NIL     <======          34     ======>         NIL
 34     <======          36     ======>          38
NIL     <======          38     ======>          39
NIL     <======          39     ======>         NIL
 36     <======          40     ======>          44
NIL     <======          41     ======>         NIL
 41     <======          42     ======>          43
NIL     <======          43     ======>         NIL
 42     <======          44     ======>          46
NIL     <======          45     ======>         NIL
 45     <======          46     ======>          47
NIL     <======          47     ======>         NIL
 40     <======          48     ======>          57
NIL     <======          49     ======>         NIL
 49     <======          50     ======>         NIL
 50     <======          53     ======>          54
NIL     <======          54     ======>          55
NIL     <======          55     ======>         NIL
 53     <======          57     ======>          60
NIL     <======          58     ======>          59
NIL     <======          59     ======>         NIL
 58     <======          60     ======>          63
NIL     <======          61     ======>         NIL
 61     <======          63     ======>         NIL
 33     <======          64     ======>          81
NIL     <======          65     ======>         NIL
 65     <======          67     ======>         NIL
 67     <======          68     ======>          70
NIL     <======          69     ======>         NIL
 69     <======          70     ======>          71
NIL     <======          71     ======>         NIL
 68     <======          72     ======>          76
NIL     <======          73     ======>         NIL
 73     <======          74     ======>          75
NIL     <======          75     ======>         NIL
 74     <======          76     ======>          78
NIL     <======          77     ======>         NIL
 77     <======          78     ======>          79
NIL     <======          79     ======>         NIL
 72     <======          81     ======>          88
NIL     <======          82     ======>         NIL
 82     <======          84     ======>          86
NIL     <======          85     ======>         NIL
 85     <======          86     ======>          87
NIL     <======          87     ======>         NIL
 84     <======          88     ======>          96
NIL     <======          89     ======>         NIL
 89     <======          90     ======>         NIL
 90     <======          92     ======>          95
NIL     <======          93     ======>         NIL
 93     <======          95     ======>         NIL
 92     <======          96     ======>          98
NIL     <======          97     ======>         NIL
 97     <======          98     ======>          99
NIL     <======          99     ======>         100
NIL     <======         100     ======>         NIL

//////////////////////////////////////////////////////////////////////
After removing 77, 66, 47, 72, 49, 70, 87, 48, 61, 86, 7, 32, 21, 62, 31, 92, 45, 94, 15,
96,
//////////////////////////////////////////////////////////////////////

        NIL     <======           1     ======>         NIL
          1     <======           3     ======>         NIL
          3     <======           5     ======>           6
        NIL     <======           6     ======>         NIL
          5     <======           8     ======>          12
```

NIL	<======	11	======>	NIL
11	<======	12	======>	14
NIL	<======	14	======>	NIL
8	<======	16	======>	24
NIL	<======	18	======>	19
NIL	<======	19	======>	NIL
18	<======	20	======>	22
NIL	<======	22	======>	23
NIL	<======	23	======>	NIL
20	<======	24	======>	28
NIL	<======	25	======>	NIL
25	<======	26	======>	27
NIL	<======	27	======>	NIL
26	<======	28	======>	30
NIL	<======	29	======>	NIL
29	<======	30	======>	NIL
16	<======	33	======>	50
NIL	<======	34	======>	NIL
34	<======	36	======>	38
NIL	<======	38	======>	39
NIL	<======	39	======>	NIL
36	<======	40	======>	44
NIL	<======	41	======>	NIL
41	<======	42	======>	43
NIL	<======	43	======>	NIL
42	<======	44	======>	46
NIL	<======	46	======>	NIL
40	<======	50	======>	57
NIL	<======	53	======>	NIL
53	<======	54	======>	55
NIL	<======	55	======>	NIL
54	<======	57	======>	60
NIL	<======	58	======>	59
NIL	<======	59	======>	NIL
58	<======	60	======>	63
NIL	<======	63	======>	NIL
33	<======	64	======>	81
NIL	<======	65	======>	NIL
65	<======	67	======>	NIL
67	<======	68	======>	71
NIL	<======	69	======>	NIL
69	<======	71	======>	NIL
68	<======	73	======>	76
NIL	<======	74	======>	75
NIL	<======	75	======>	NIL
74	<======	76	======>	78
NIL	<======	78	======>	79
NIL	<======	79	======>	NIL
73	<======	81	======>	93
NIL	<======	82	======>	NIL
82	<======	84	======>	85
NIL	<======	85	======>	NIL
84	<======	88	======>	90
NIL	<======	89	======>	NIL
89	<======	90	======>	NIL
88	<======	93	======>	97
NIL	<======	95	======>	NIL
95	<======	97	======>	99
NIL	<======	98	======>	NIL
98	<======	99	======>	100
NIL	<======	100	======>	NIL

//
After removing 49, 14, 51, 4, 85, 78, 43, 96, 85, 38, 71, 0, 33, 58, 67, 56, 65, 62, 55, 20,
//

NIL	<======	1	======>	NIL
1	<======	3	======>	NIL
3	<======	5	======>	6

```
NIL     <======          6     ======>         NIL
  5     <======          8     ======>          12
NIL     <======         11     ======>         NIL
 11     <======         12     ======>         NIL
  8     <======         16     ======>          24
NIL     <======         18     ======>          19
NIL     <======         19     ======>         NIL
 18     <======         22     ======>          23
NIL     <======         23     ======>         NIL
 22     <======         24     ======>          28
NIL     <======         25     ======>         NIL
 25     <======         26     ======>          27
NIL     <======         27     ======>         NIL
 26     <======         28     ======>          30
NIL     <======         29     ======>         NIL
 29     <======         30     ======>         NIL
 16     <======         34     ======>          50
NIL     <======         36     ======>          39
NIL     <======         39     ======>         NIL
 36     <======         40     ======>          44
NIL     <======         41     ======>         NIL
 41     <======         42     ======>         NIL
 42     <======         44     ======>          46
NIL     <======         46     ======>         NIL
 40     <======         50     ======>          57
NIL     <======         53     ======>         NIL
 53     <======         54     ======>         NIL
 54     <======         57     ======>          60
NIL     <======         59     ======>         NIL
 59     <======         60     ======>          63
NIL     <======         63     ======>         NIL
 34     <======         64     ======>          81
NIL     <======         68     ======>          69
NIL     <======         69     ======>         NIL
 68     <======         73     ======>          76
NIL     <======         74     ======>          75
NIL     <======         75     ======>         NIL
 74     <======         76     ======>          79
NIL     <======         79     ======>         NIL
 73     <======         81     ======>          93
NIL     <======         82     ======>         NIL
 82     <======         84     ======>         NIL
 84     <======         88     ======>          90
NIL     <======         89     ======>         NIL
 89     <======         90     ======>         NIL
 88     <======         93     ======>          97
NIL     <======         95     ======>         NIL
 95     <======         97     ======>          99
NIL     <======         98     ======>         NIL
 98     <======         99     ======>         100
NIL     <======        100     ======>         NIL
```

```
////////////////////////////////////////////////////////////////////////
After removing 45, 2, 83, 64, 29, 18, 99, 40, 57, 50, 63, 36, 45, 54, 3, 0, 89, 62, 59,
24,
////////////////////////////////////////////////////////////////////////
```

```
NIL     <======          1     ======>         NIL
  1     <======          5     ======>           6
NIL     <======          6     ======>         NIL
  5     <======          8     ======>          12
NIL     <======         11     ======>         NIL
 11     <======         12     ======>         NIL
  8     <======         16     ======>          22
NIL     <======         19     ======>         NIL
 19     <======         22     ======>          23
NIL     <======         23     ======>         NIL
 16     <======         25     ======>          34
NIL     <======         26     ======>          27
NIL     <======         27     ======>         NIL
```

```
 26     <======          28     ======>          30
NIL     <======          30     ======>         NIL
 28     <======          34     ======>          44
NIL     <======          39     ======>         NIL
 39     <======          41     ======>          42
NIL     <======          42     ======>         NIL
 41     <======          44     ======>          53
NIL     <======          46     ======>         NIL
 46     <======          53     ======>          60
NIL     <======          60     ======>         NIL
 25     <======          68     ======>          81
NIL     <======          69     ======>         NIL
 69     <======          73     ======>         NIL
 73     <======          74     ======>          76
NIL     <======          75     ======>         NIL
 75     <======          76     ======>          79
NIL     <======          79     ======>         NIL
 74     <======          81     ======>          93
NIL     <======          82     ======>         NIL
 82     <======          84     ======>         NIL
 84     <======          88     ======>          90
NIL     <======          90     ======>         NIL
 88     <======          93     ======>          97
NIL     <======          95     ======>         NIL
 95     <======          97     ======>          100
NIL     <======          98     ======>         NIL
 98     <======         100     ======>         NIL

////////////////////////////////////////////////////////////////////////
After removing 97, 70, 59, 76, 1, 74, 47, 8, 1, 54, 7, 36, 85, 98, 23, 52, 97, 70, 11,
44,
////////////////////////////////////////////////////////////////////////

NIL     <======           5     ======>         NIL
  5     <======           6     ======>          12
NIL     <======          12     ======>         NIL
  6     <======          16     ======>          22
NIL     <======          19     ======>         NIL
 19     <======          22     ======>         NIL
 16     <======          25     ======>          34
NIL     <======          26     ======>          27
NIL     <======          27     ======>         NIL
 26     <======          28     ======>          30
NIL     <======          30     ======>         NIL
 28     <======          34     ======>          46
NIL     <======          39     ======>         NIL
 39     <======          41     ======>          42
NIL     <======          42     ======>         NIL
 41     <======          46     ======>          53
NIL     <======          53     ======>          60
NIL     <======          60     ======>         NIL
 25     <======          68     ======>          81
NIL     <======          69     ======>         NIL
 69     <======          73     ======>         NIL
 73     <======          75     ======>          79
NIL     <======          79     ======>         NIL
 75     <======          81     ======>          93
NIL     <======          82     ======>         NIL
 82     <======          84     ======>         NIL
 84     <======          88     ======>          90
NIL     <======          90     ======>         NIL
 88     <======          93     ======>          100
NIL     <======          95     ======>         NIL
 95     <======         100     ======>         NIL

////////////////////////////////////////////////////////////////////////
After removing 33, 98, 35, 36, 33, 30, 95, 48, 77, 46, 55, 40, 61, 90, 23, 44, 69, 30,
19, 12,
////////////////////////////////////////////////////////////////////////
```

```
NIL     <======          5     ======>        NIL
  5     <======          6     ======>        NIL
  6     <======         16     ======>         22
NIL     <======         22     ======>        NIL
 16     <======         25     ======>         34
NIL     <======         26     ======>        NIL
 26     <======         27     ======>         28
NIL     <======         28     ======>        NIL
 27     <======         34     ======>         53
NIL     <======         39     ======>        NIL
 39     <======         41     ======>         42
NIL     <======         42     ======>        NIL
 41     <======         53     ======>         60
NIL     <======         60     ======>        NIL
 25     <======         68     ======>         81
NIL     <======         73     ======>        NIL
 73     <======         75     ======>         79
NIL     <======         79     ======>        NIL
 75     <======         81     ======>         88
NIL     <======         82     ======>        NIL
 82     <======         84     ======>        NIL
 84     <======         88     ======>         93
NIL     <======         93     ======>        100
NIL     <======        100     ======>        NIL
```

//
After removing 89, 6, 7, 84, 17, 26, 59, 0, 77, 46, 71, 76, 21, 74, 51, 52, 93, 62, 67, 48,
//

```
NIL     <======          5     ======>        NIL
  5     <======         16     ======>         22
NIL     <======         22     ======>        NIL
 16     <======         25     ======>         27
NIL     <======         27     ======>         28
NIL     <======         28     ======>        NIL
 25     <======         34     ======>         53
NIL     <======         39     ======>        NIL
 39     <======         41     ======>         42
NIL     <======         42     ======>        NIL
 41     <======         53     ======>         60
NIL     <======         60     ======>        NIL
 34     <======         68     ======>         81
NIL     <======         73     ======>        NIL
 73     <======         75     ======>         79
NIL     <======         79     ======>        NIL
 75     <======         81     ======>         88
NIL     <======         82     ======>        NIL
 82     <======         88     ======>        100
NIL     <======        100     ======>        NIL
```

//
After removing 17, 82, 35, 92, 53, 70, 19, 56, 13, 26, 39, 40, 1, 62, 7, 12, 93, 86, 95, 44,
//

```
NIL     <======          5     ======>        NIL
  5     <======         16     ======>         22
NIL     <======         22     ======>        NIL
 16     <======         25     ======>         27
NIL     <======         27     ======>         28
NIL     <======         28     ======>        NIL
 25     <======         34     ======>         42
NIL     <======         41     ======>        NIL
 41     <======         42     ======>         60
NIL     <======         60     ======>        NIL
 34     <======         68     ======>         81
NIL     <======         73     ======>        NIL
```

```
 73     <======          75      =======>         79
NIL     <======          79      =======>        NIL
 75     <======          81      =======>         88
NIL     <======          88      =======>        100
NIL     <======         100      =======>        NIL
```

```
////////////////////////////////////////////////////////////////////////
After removing 33, 94, 23, 28, 53, 14, 43, 28, 45, 58, 39, 56, 5, 78, 47, 64, 61, 2, 31,
88,
////////////////////////////////////////////////////////////////////////
```

```
NIL     <======          16      =======>         22
NIL     <======          22      =======>        NIL
 16     <======          25      =======>         27
NIL     <======          27      =======>        NIL
 25     <======          34      =======>         42
NIL     <======          41      =======>        NIL
 41     <======          42      =======>         60
NIL     <======          60      =======>        NIL
 34     <======          68      =======>         81
NIL     <======          73      =======>        NIL
 73     <======          75      =======>         79
NIL     <======          79      =======>        NIL
 75     <======          81      =======>        100
NIL     <======         100      =======>        NIL
```

```
////////////////////////////////////////////////////////////////////////
// Here is a height balance tree with numbers 30 to 1 (left rotation):
////////////////////////////////////////////////////////////////////////
```

```
NIL     <======           1      =======>        NIL
  1     <======           2      =======>          3
NIL     <======           3      =======>        NIL
  2     <======           4      =======>          5
NIL     <======           5      =======>        NIL
  4     <======           6      =======>         10
NIL     <======           7      =======>        NIL
  7     <======           8      =======>          9
NIL     <======           9      =======>        NIL
  8     <======          10      =======>         12
NIL     <======          11      =======>        NIL
 11     <======          12      =======>         13
NIL     <======          13      =======>        NIL
  6     <======          14      =======>         22
NIL     <======          15      =======>        NIL
 15     <======          16      =======>         17
NIL     <======          17      =======>        NIL
 16     <======          18      =======>         20
NIL     <======          19      =======>        NIL
 19     <======          20      =======>         21
NIL     <======          21      =======>        NIL
 18     <======          22      =======>         25
NIL     <======          23      =======>        NIL
 23     <======          24      =======>        NIL
 24     <======          25      =======>         27
NIL     <======          26      =======>        NIL
 26     <======          27      =======>         28
NIL     <======          28      =======>        NIL
 14     <======          29      =======>         73
NIL     <======          30      =======>        NIL
 30     <======          41      =======>         42
NIL     <======          42      =======>         60
NIL     <======          60      =======>        NIL
 41     <======          73      =======>         81
NIL     <======          75      =======>         79
NIL     <======          79      =======>        NIL
 75     <======          81      =======>        100
NIL     <======         100      =======>        NIL
```

```
////////////////////////////////////////////////////////////////////////
// Here is a height balance tree with double rotation
////////////////////////////////////////////////////////////////////////

        NIL     <======          5     ======>         NIL
          5     <======          9     ======>          10
        NIL     <======         10     ======>         NIL
          9     <======         30     ======>          51
        NIL     <======         50     ======>         NIL
         50     <======         51     ======>          53
        NIL     <======         52     ======>         NIL
         52     <======         53     ======>          55
        NIL     <======         55     ======>         NIL


////////////////////////////////////////////////////////////////////////
// Here is a data structure with 10 random numbers:
////////////////////////////////////////////////////////////////////////

        NIL     <======         35     ======>         NIL
         35     <======        604     ======>         NIL
        604     <======        810     ======>        6152
        NIL     <======       1213     ======>         NIL
       1213     <======       3956     ======>        6102
        NIL     <======       6102     ======>         NIL
       3956     <======       6152     ======>        9221
        NIL     <======       6383     ======>         NIL
       6383     <======       9221     ======>        9609
        NIL     <======       9609     ======>         NIL


////////////////////////////////////////////////////////////////////////
// The following data structure is a copy of the above one:
////////////////////////////////////////////////////////////////////////

        NIL     <======         35     ======>         NIL
         35     <======        604     ======>         NIL
        604     <======        810     ======>        6152
        NIL     <======       1213     ======>         NIL
       1213     <======       3956     ======>        6102
        NIL     <======       6102     ======>         NIL
       3956     <======       6152     ======>        9221
        NIL     <======       6383     ======>         NIL
       6383     <======       9221     ======>        9609
        NIL     <======       9609     ======>         NIL


////////////////////////////////////////////////////////////////////////
// After inserting 12345, 4321, 15, 8876, 5034:
////////////////////////////////////////////////////////////////////////

        NIL     <======         15     ======>         NIL
         15     <======         35     ======>         604
        NIL     <======        604     ======>         NIL
         35     <======        810     ======>        3956
        NIL     <======       1213     ======>         NIL
       1213     <======       3956     ======>        5034
        NIL     <======       4321     ======>         NIL
       4321     <======       5034     ======>        6102
        NIL     <======       6102     ======>         NIL
        810     <======       6152     ======>        9221
        NIL     <======       6383     ======>        8876
        NIL     <======       8876     ======>         NIL
       6383     <======       9221     ======>        9609
        NIL     <======       9609     ======>       12345
        NIL     <======      12345     ======>         NIL


////////////////////////////////////////////////////////////////////////
// Search for the newly inserted numbers
```

```
////////////////////////////////////////////////////////////////////////

Find number 12345
Find number 4321
Find number 15
Find number 8876
Find number 5034
Cannot find random number 7762


////////////////////////////////////////////////////////////////////////
// Remove test
////////////////////////////////////////////////////////////////////////

===After removing 12345, 4321, 15,

         NIL     <======          35      ======>         604
         NIL     <======         604      ======>         NIL
          35     <======         810      ======>        3956
         NIL     <======        1213      ======>         NIL
        1213     <======        3956      ======>        5034
         NIL     <======        5034      ======>        6102
         NIL     <======        6102      ======>         NIL
         810     <======        6152      ======>        9221
         NIL     <======        6383      ======>        8876
         NIL     <======        8876      ======>         NIL
        6383     <======        9221      ======>        9609
         NIL     <======        9609      ======>         NIL

===After removing 8876, 5034,

         NIL     <======          35      ======>         604
         NIL     <======         604      ======>         NIL
          35     <======         810      ======>        1213
         NIL     <======        1213      ======>         NIL
         810     <======        3956      ======>        6152
         NIL     <======        6102      ======>         NIL
        6102     <======        6152      ======>        9221
         NIL     <======        6383      ======>         NIL
        6383     <======        9221      ======>        9609
         NIL     <======        9609      ======>         NIL


////////////////////////////////////////////////////////////////////////
// Remove a random number from the above structure.
////////////////////////////////////////////////////////////////////////

Cannot remove 7762


////////////////////////////////////////////////////////////////////////
// After empty(), the above data structure become:
////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////
// Measure time needed to build and access the data structure.
////////////////////////////////////////////////////////////////////////

====== Data structure with 10000 nodes ======
Time to build the data structure: 621 milliseconds
Time to access the data structure for 10000 times: 80 milliseconds
Space required to build the data structure: 200008 bytes

====== Data structure with 30000 nodes ======
Time to build the data structure: 2013 milliseconds
Time to access the data structure for 30000 times: 300 milliseconds
Space required to build the data structure: 600008 bytes

====== Data structure with 50000 nodes ======
```

```
Time to build the data structure: 3876 milliseconds
Time to access the data structure for 50000 times: 551 milliseconds
Space required to build the data structure: 1000008 bytes

====== Data structure with 70000 nodes ======
Time to build the data structure: 5268 milliseconds
Time to access the data structure for 70000 times: 811 milliseconds
Space required to build the data structure: 1400008 bytes

====== Data structure with 90000 nodes ======
Time to build the data structure: 6990 milliseconds
Time to access the data structure for 90000 times: 1102 milliseconds
Space required to build the data structure: 1799988 bytes

====== Data structure with 110000 nodes ======
Time to build the data structure: 8733 milliseconds
Time to access the data structure for 110000 times: 1382 milliseconds
Space required to build the data structure: 2200008 bytes

====== Data structure with 130000 nodes ======
Time to build the data structure: 10445 milliseconds
Time to access the data structure for 130000 times: 1652 milliseconds
Space required to build the data structure: 2599968 bytes

====== Data structure with 150000 nodes ======
Time to build the data structure: 12268 milliseconds
Time to access the data structure for 150000 times: 1952 milliseconds
Space required to build the data structure: 2999948 bytes

====== Data structure with 170000 nodes ======
Time to build the data structure: 14111 milliseconds
Time to access the data structure for 170000 times: 2253 milliseconds
Space required to build the data structure: 3399948 bytes

====== Data structure with 190000 nodes ======
Time to build the data structure: 15933 milliseconds
Time to access the data structure for 190000 times: 2584 milliseconds
Space required to build the data structure: 3799828 bytes

====== Data structure with 210000 nodes ======
Time to build the data structure: 17836 milliseconds
Time to access the data structure for 210000 times: 2904 milliseconds
Space required to build the data structure: 4199868 bytes

====== Data structure with 230000 nodes ======
Time to build the data structure: 19649 milliseconds
Time to access the data structure for 230000 times: 3214 milliseconds
Space required to build the data structure: 4599948 bytes

====== Data structure with 250000 nodes ======
Time to build the data structure: 21541 milliseconds
Time to access the data structure for 250000 times: 3505 milliseconds
Space required to build the data structure: 4999908 bytes

====== Data structure with 270000 nodes ======
Time to build the data structure: 23443 milliseconds
Time to access the data structure for 270000 times: 3836 milliseconds
Space required to build the data structure: 5399868 bytes

====== Data structure with 290000 nodes ======
Time to build the data structure: 25376 milliseconds
Time to access the data structure for 290000 times: 4217 milliseconds
Space required to build the data structure: 5799748 bytes

====== Data structure with 310000 nodes ======
Time to build the data structure: 27179 milliseconds
Time to access the data structure for 310000 times: 4466 milliseconds
Space required to build the data structure: 6199788 bytes
```

Test Result 7: *TestSList.txt*

```
================================================
 Test Singly Linked List Built With Null Node Object
================================================


//////////////////////////////////////////////////////////////////////
// Test insert(), append() and remove()
//////////////////////////////////////////////////////////////////////


After inserting numbers 1 to 10:
10      9               8               7
6       5               4               3
2       1

After appending numbers 1 to 10:
10      9               8               7
6       5               4               3
2       1               1               2
3       4               5               6
7       8               9               10

After removing 1:
10      9               8               7
6       5               4               3
2       2               3               4
5       6               7               8
9       10
Number of nodes removed: 2

After removing 2:
10      9               8               7
6       5               4               3
3       4               5               6
7       8               9               10

Number of nodes removed: 2

After removing 11:
10      9               8               7
6       5               4               3
3       4               5               6
7       8               9               10

Number of nodes removed: 0


//////////////////////////////////////////////////////////////////////
// Insert five 2s and remove 2:
//////////////////////////////////////////////////////////////////////


After insert five 2s:
2       2               2               2
2
After removing 2:

Number of nodes removed: 5


//////////////////////////////////////////////////////////////////////
// Here is a data structure with 10 random numbers:
//////////////////////////////////////////////////////////////////////

9221            3956            6383            6102
9609            6152            35              810
1213            604
```

```
////////////////////////////////////////////////////////////////////////
// The following data structure is a copy of the above one:
////////////////////////////////////////////////////////////////////////

9221            3956            6383            6102
9609            6152            35              810
1213            604

////////////////////////////////////////////////////////////////////////
// After inserting 12345, 4321, 15, 8876, 5034:
////////////////////////////////////////////////////////////////////////

5034            8876            15              4321
12345           9221            3956            6383
6102            9609            6152            35
810             1213            604

////////////////////////////////////////////////////////////////////////
// Search for the newly inserted numbers
////////////////////////////////////////////////////////////////////////

Find number 12345
Find number 4321
Find number 15
Find number 8876
Find number 5034
Cannot find random number 7762


////////////////////////////////////////////////////////////////////////
// Remove test
////////////////////////////////////////////////////////////////////////

===After removing 12345, 4321, 15,

5034            8876            9221            3956
6383            6102            9609            6152
35      810             1213            604

===After removing 8876, 5034,

9221            3956            6383            6102
9609            6152            35              810
1213            604

////////////////////////////////////////////////////////////////////////
// Remove a random number from the above structure.
////////////////////////////////////////////////////////////////////////

Cannot remove 7762


////////////////////////////////////////////////////////////////////////
// After empty(), the above data structure become:
////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////
// Measure time needed to build and access the data structure.
////////////////////////////////////////////////////////////////////////

====== Data structure with 10000 nodes ======
Time to build the data structure: 31 milliseconds
Time to access the data structure for 10000 times: 3004 milliseconds
Space required to build the data structure: 120012 bytes

====== Data structure with 20000 nodes ======
Time to build the data structure: 70 milliseconds
Time to access the data structure for 20000 times: 6569 milliseconds
```

```
Space required to build the data structure: 240008 bytes

====== Data structure with 30000 nodes ======
Time to build the data structure: 110 milliseconds
Time to access the data structure for 30000 times: 10154 milliseconds
Space required to build the data structure: 360008 bytes

====== Data structure with 40000 nodes ======
Time to build the data structure: 151 milliseconds
Time to access the data structure for 40000 times: 13609 milliseconds
Space required to build the data structure: 480008 bytes

====== Data structure with 50000 nodes ======
Time to build the data structure: 180 milliseconds
Time to access the data structure for 50000 times: 16985 milliseconds
Space required to build the data structure: 600008 bytes

====== Data structure with 60000 nodes ======
Time to build the data structure: 220 milliseconds
Time to access the data structure for 60000 times: 20380 milliseconds
Space required to build the data structure: 720008 bytes

====== Data structure with 70000 nodes ======
Time to build the data structure: 260 milliseconds
Time to access the data structure for 70000 times: 23764 milliseconds
Space required to build the data structure: 840008 bytes

====== Data structure with 80000 nodes ======
Time to build the data structure: 300 milliseconds
Time to access the data structure for 80000 times: 27159 milliseconds
Space required to build the data structure: 960008 bytes

====== Data structure with 90000 nodes ======
Time to build the data structure: 331 milliseconds
Time to access the data structure for 90000 times: 30554 milliseconds
Space required to build the data structure: 1080008 bytes

====== Data structure with 100000 nodes ======
Time to build the data structure: 360 milliseconds
Time to access the data structure for 100000 times: 33959 milliseconds
Space required to build the data structure: 1200008 bytes
```

Test Result 8: *TestPtrSList.txt*

```
================================================
 Test Singly Linked List Built With NULL pointers
================================================

//////////////////////////////////////////////////////////////////////
// Test insert(), append() and remove()
//////////////////////////////////////////////////////////////////////

After inserting numbers 1 to 10:
10      9               8               7
6       5               4               3
2       1

After appending numbers 1 to 10:
10      9               8               7
6       5               4               3
2       1               1               2
3       4               5               6
7       8               9               10

After removing 1:
10      9               8               7
6       5               4               3
2       2               3               4
5       6               7               8
9       10
Number of nodes removed: 2

After removing 2:
10      9               8               7
6       5               4               3
3       4               5               6
7       8               9               10

Number of nodes removed: 2

After removing 11:
10      9               8               7
6       5               4               3
3       4               5               6
7       8               9               10

Number of nodes removed: 0

//////////////////////////////////////////////////////////////////////
// Insert five 2s and remove 2:
//////////////////////////////////////////////////////////////////////

After insert five 2s:
2       2               2               2
2
After removing 2:

Number of nodes removed: 5

//////////////////////////////////////////////////////////////////////
// Here is a data structure with 10 random numbers:
//////////////////////////////////////////////////////////////////////

9221            3956            6383            6102
9609            6152            35              810
1213            604
```

```
////////////////////////////////////////////////////////////////////////
// The following data structure is a copy of the above one:
////////////////////////////////////////////////////////////////////////

9221            3956            6383            6102
9609            6152            35              810
1213            604

////////////////////////////////////////////////////////////////////////
// After inserting 12345, 4321, 15, 8876, 5034:
////////////////////////////////////////////////////////////////////////

5034            8876            15              4321
12345           9221            3956            6383
6102            9609            6152            35
810             1213            604

////////////////////////////////////////////////////////////////////////
// Search for the newly inserted numbers
////////////////////////////////////////////////////////////////////////

Find number 12345
Find number 4321
Find number 15
Find number 8876
Find number 5034
Cannot find random number 7762


////////////////////////////////////////////////////////////////////////
// Remove test
////////////////////////////////////////////////////////////////////////

===After removing 12345, 4321, 15,

5034            8876            9221            3956
6383            6102            9609            6152
35      810             1213            604

===After removing 8876, 5034,

9221            3956            6383            6102
9609            6152            35              810
1213            604

////////////////////////////////////////////////////////////////////////
// Remove a random number from the above structure.
////////////////////////////////////////////////////////////////////////

Cannot remove 7762


////////////////////////////////////////////////////////////////////////
// After empty(), the above data structure become:
////////////////////////////////////////////////////////////////////////



////////////////////////////////////////////////////////////////////////
// Measure time needed to build and access the data structure.
////////////////////////////////////////////////////////////////////////

====== Data structure with 10000 nodes ======
Time to build the data structure: 40 milliseconds
Time to access the data structure for 10000 times: 1842 milliseconds
Space required to build the data structure: 120008 bytes

====== Data structure with 20000 nodes ======
Time to build the data structure: 70 milliseconds
Time to access the data structure for 20000 times: 4267 milliseconds
```

```
Space required to build the data structure: 240008 bytes

====== Data structure with 30000 nodes ======
Time to build the data structure: 100 milliseconds
Time to access the data structure for 30000 times: 6749 milliseconds
Space required to build the data structure: 360008 bytes

====== Data structure with 40000 nodes ======
Time to build the data structure: 130 milliseconds
Time to access the data structure for 40000 times: 9183 milliseconds
Space required to build the data structure: 480008 bytes

====== Data structure with 50000 nodes ======
Time to build the data structure: 170 milliseconds
Time to access the data structure for 50000 times: 11437 milliseconds
Space required to build the data structure: 600008 bytes

====== Data structure with 60000 nodes ======
Time to build the data structure: 200 milliseconds
Time to access the data structure for 60000 times: 13730 milliseconds
Space required to build the data structure: 720008 bytes

====== Data structure with 70000 nodes ======
Time to build the data structure: 240 milliseconds
Time to access the data structure for 70000 times: 16003 milliseconds
Space required to build the data structure: 840008 bytes

====== Data structure with 80000 nodes ======
Time to build the data structure: 270 milliseconds
Time to access the data structure for 80000 times: 18296 milliseconds
Space required to build the data structure: 960008 bytes

====== Data structure with 90000 nodes ======
Time to build the data structure: 300 milliseconds
Time to access the data structure for 90000 times: 20590 milliseconds
Space required to build the data structure: 1080008 bytes

====== Data structure with 100000 nodes ======
Time to build the data structure: 340 milliseconds
Time to access the data structure for 100000 times: 22843 milliseconds
Space required to build the data structure: 1200008 bytes
```